Université de Montréal

Identification d'une architecture à base de composants dans une application orientée objets à l'aide d'une analyse dynamique

par
Simon Allier

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

novembre, 2011

Université de Montréal
Faculté des études supérieures

Cette thèse intitulée:

Identification d'une architecture à base de composants dans une application orientée objets à l'aide d'une analyse dynamique

présentée par:

Simon Allier

a été évaluée par un jury composé des personnes suivantes:

Falvio Oquendo,	président-rapporteur
Houari Sahraoui,	directeur de recherche
Salah Sadou,	codirecteur
Antoine Beugnard,	membre du jury
Bruno Dufour,	examinateur externe

Thèse acceptée le: 02 juin 2012

Un système, décrit avec un grand nombre d'éléments fortement interdépendants, est complexe, difficile à comprendre et à maintenir. Ainsi, une application orientée objet est souvent complexe, car elle contient des centaines de classes avec de nombreuses dépendances plus ou moins explicites. Une même application, utilisant le paradigme composant, contiendrait un plus petit nombre d'éléments, faiblement couplés entre eux et avec des interdépendances clairement définies. Ceci est dû au fait que le paradigme composant fournit une bonne représentation de haut niveau des systèmes complexes. Ainsi, ce paradigme peut être utilisé comme "espace de projection" des systèmes orientés objets. Une telle projection peut faciliter l'étape de compréhension d'un système, un prérequis nécessaire avant toute activité de maintenance et/ou d'évolution.

De plus, il est possible d'utiliser cette représentation, comme un modèle pour effectuer une restructuration complète d'une application orientée objets opérationnelle vers une application équivalente à base de composants tout aussi opérationnelle. Ainsi, La nouvelle application bénéficiant ainsi, de toutes les bonnes propriétés associées au paradigme composants.

L'objectif de ma thèse est de proposer une méthode semi-automatique pour identifier une architecture à base de composants dans une application orientée objets. Cette architecture doit, non seulement aider à la compréhension de l'application originale, mais aussi simplifier la projection de cette dernière dans un modèle concret de composant. L'identification d'une architecture à base de composants est réalisée en trois grandes étapes : i) obtention des données nécessaires au processus d'identification. Elles correspondent aux dépendances entre les classes et sont obtenues avec une analyse dynamique de l'application cible. ii) identification des composants. Trois méthodes ont été explorées. La première utilise un treillis de Galois, la seconde deux méta-heuristiques et la dernière une méta-heuristique multi-objective. iii) identification de l'architecture à base de composants de l'application cible. Cela est fait en identifiant les interfaces requises et fournis pour chaque composant.

Afin de valider ce processus d'identification, ainsi que les différents choix faits durant son développement, j'ai réalisé différentes études de cas. Enfin, je montre la faisabilité de la projection de l'architecture à base de composants identifiée vers un modèle

concret de composants.

ABSTRACT

A system is complex and particularly difficult to understand and to maintain when it is described with a large number of highly interdependent parties. An object-oriented application is often complex because it uses hundreds or thousands of classes with many different dependencies more or less explicit. The same application, using the component paradigm, contains a smaller number of loosely coupled parties, highly cohesive with clear inter-dependencies. Indeed, because the component paradigm provides a high-level representation, synthetic and well-organized structure of complex systems, it can provide a space of projection for object-oriented applications. Such projection facilitates the step of understanding a system prior to any activity of maintenance and/or evolution.

In addition, it is possible to use this representation as a model to perform a complete restructuring of an operational object-oriented application into its equivalent operational component-based application. Thus, the new form of the application benefits from all the good properties associated with the component-oriented paradigm.

The goal of my thesis is to propose a semi-automatic approach to identify a component-based architecture in an object-oriented application. This architecture should help in understanding the original application, but also simplifies the projection of the object-oriented application on a concrete component model.

The identification of a component-based architecture is achieved in three main steps : i) obtaining data for the identification process. These data, which correspond to dependencies between classes, are obtained with a dynamic analysis of the target application. ii) identification of the components. Three methods were explored. The first uses the formal concept analysis, the second two meta-heuristics and the last a multiobjective meta-heuristic. iii) identification of the component-based architecture representing the target application. This is done by identifying the provided and required interfaces for each component.

To validate this identification process, and the different choices made during its development, I realized several case studies. Finally, I show the feasibility of the projection of the identified component-based architecture on a specific component model.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

Première partie

Introduction

CHAPITRE 1

INTRODUCTION

1.1 Motivation

Une caractéristique fondamentale des logiciels est le besoin d'évoluer pour répondre à de nouvelles spécifications. La première loi de Lehman [LB85] illustre ce fait : *Un programme utilisé dans un environnement du monde réel doit nécessairement changer sinon il deviendra progressivement de moins en moins utile dans cet environnement*. Bien qu'énoncée dans les années 70, cette loi n'a jamais été contredite. De plus, le coût de la maintenance d'un logiciel est en constante augmentation : alors que les études faites dans les années 80 montraient que la part de la maintenance représentait entre 50% et 60% du coût global [LS81, McK84], des études plus récentes portent cette proportion entre 80% et 90% [Erl00, SPL03]. En outre, il est important de noter que parmi les différentes étapes de la maintenance, la compréhension de l'application, pré-requis nécessaire avant toute activité de maintenance et/ou d'évolution, représente plus de 50% du coût total de la maintenance [BR00].

Or, le paradigme objet majoritairement utilisé de nos jours, produit des applications complexes, difficile à comprendre et à maintenir. En effet, les systèmes orientés objets[1] contiennent des centaines voir des milliers de classes, fortement interdépendantes et liées par des dépendances plus ou moins explicites telles que l'héritage ou les appels de méthodes. De plus, la documentation d'un système, qui a pour principal but l'aide à la compréhension de celui-ci, n'est pas toujours complète, ou bien n'est pas mise à jour durant l'évolution de l'application [Par00, VC02]. Ainsi, la compréhension d'une application OO peut être très difficile, longue et sujette à diverses interprétations. Cela rend coûteux et difficile l'évolution et la maintenance de ces applications.

[1] par la suite, orienté objet sera noté OO

Par ailleurs, une même application, utilisant le paradigme composant, contiendrait un plus petit nombre d'éléments, des *composants*, faiblement couplés entre eux et avec des interdépendances clairement définies à travers leur *interfaces* fournies et requises. La composition de ces composants, suivant une architecture, forme une application à base de composant. De ce fait, il est clairement établi que le paradigme composant fournit une bonne représentation de haut niveau des systèmes complexes.

Ainsi, le paradigme composant peut fournir un "espace de projection" qui simplifie la compréhension d'un système OO. Dans ce cas, le paradigme composant est utilisé comme un outil de documentation pour améliorer la compréhension de l'application OO. En effet, il permet la construction d'une architecture plus simple que celle fournie par un système OO : cette architecture contient moins d'éléments et leurs dépendances clairement définies. De plus, un composant identifié dans un système OO est typiquement un artefact avec un haut niveau d'abstraction, ne représentant pas une unique classe, mais le plus souvent un ensemble de classes fournissant une fonctionnalité de haut niveau. Ainsi, l'identification d'une telle architecture dans un système OO peut grandement faciliter les étapes de maintenance et d'évolution.

L'identification de nouveaux concepts dans un code n'est pas une nouvelle idée. De nombreuses approches ont déjà été proposées tel que l'identification de classes dans un code procédural [LW90, GK95, CCM96, SMLD97]. Des approches pour identifier des composants [WF05] ou une architecture à base de composants [MM08, CKK08] dans une application ont déjà été proposées. Ces approches utilisent toutes le même schéma général pour identifier l'architecture à base de composant de l'application cible de ce processus (voir Figure 1.1) :

1. Récupération des données nécessaires au processus d'identification : les relations et dépendances entre les classes de l'application cible, obtenues à partir des différents artefacts de l'application cible. L'ensemble des classes/relations peut être représenté sous la forme d'un graphe où les noeuds sont des classes et les arcs les relations entre ces classes.

2. Identification des composants : un algorithme est appliqué à ce graphe afin d'obtenir une partition de l'ensemble des classes. Chaque sous-ensemble de classes de la partition est un des composants identifiés. L'identification de cette partition est guidée par différent critères tel qu'une forte cohésion dans les composants et un couplage faible entre les composants.

3. Identification de l'architecture à base de composants de l'application cible ; cela se fait le plus souvent en identifiant les interfaces des composants. Cette étape est rarement réalisée dans les approches actuelles.

Au final, les composants identifiés et leurs relations à travers leurs interfaces forment l'architecture à base de composants[2] du système cible.

Cette ABC, bien qu'utile pour la compréhension, n'est qu'une représentation "contemplative"utilisable uniquement par le concepteur. En effet, aucun élément concret de cette ABC n'est créé ou instancié. Or, il est possible d'utiliser cette représentation, comme un modèle pour effectuer une restructuration complète d'une application OO opérationnelle vers une application équivalente à base de composants tout aussi opérationnelle. Cela revient à projeter l'ABC identifiée et "abstraite" vers un modèle concret de composants. La nouvelle application bénéficiant ainsi, de toutes les bonnes propriétés associées au paradigme composant.

En effet, le processus d'identification des composants est toujours guidé par certains critères de qualité associés à ce paradigme : l'implémentation d'une fonctionnalité de haut niveau, l'optimisation de certaines métriques structurelles, etc. Ainsi, l'ABC formée à partir des composants identifiés et les interactions entre leurs interfaces fournies et requises a de meilleures propriétés structurelles que l'application OO originale.

Ainsi, l'objectif de cette thèse est de proposer une méthode semi-automatique pour identifier une architecture à base de composants. Cette architecture doit, non seulement

[2]par la suite, architecture à base de composants sera généralement noté ABC, et qu'il ne faut pas confondre avec application à basé de composants

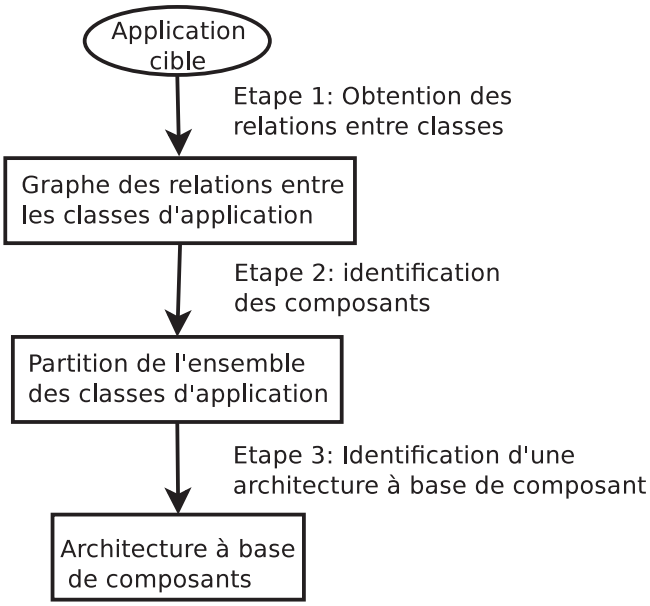


Figure 1.1 – Identification d'une architecture à base de composants

aider à la compréhension de l'application originale, mais aussi simplifier la projection de cette dernière dans un modèle concret de composants

1.2 Problématiques

Comme nous venons de le voir, il existe déjà des approches pour l'identification des composants, puis d'une architecture à base de composants dans une application orientée objet. Mais il reste de nombreux problèmes à résoudre à chaque étape du processus d'identification d'une ABC.

1.2.1 Obtention des relations entre classes

Pour déterminer les relations entre les classes, toutes les approches existantes utilisent la documentation et/ou le code source. Or, ces artefacts posent des problèmes.

Une enquête sur des logiciels industriels réalisée en 2002 a révélé que la satisfaction à l'égard de la qualité des documents est faible, voir très faible dans 84% des projets étudiés [VC02]. De même, de nombreuses études pointent la qualité et complétude de la documentation comme un problème majeur dans l'industrie, plus précisément, durant

la phase de maintenance d'une application [LS, Cha85, Sou98, VC02]. Ainsi, utiliser la documentation comme donnée d'entrée pour identifier une ABC dans le but d'aider à la compréhension me semble être une erreur, voire même contre-productif. De plus, si la documentation n'est pas mise à jour, l'ABC obtenue représentera une version antérieur de cette application. Si la documentation n'est pas complète, l'ABC obtenu risque d'être fausse.

Un autre problème concerne l'obtention statique des dépendances. En effet, certaines dépendances sont difficiles à obtenir précisément avec une analyse statique simple. En effet, si l'héritage ou la redéfinition de méthodes sont faciles à obtenir avec une simple analyse statique, certaines caractéristiques telles que le polymorphisme, le chargement dynamique de classe ou encore la réflexion ne sont pas faciles à simuler avec une simple analyse statique [ZD08]. Sans analyse statique précise du code, des liens tels que les appels entre méthodes sont omis, ou au contraire surévalués.

De plus, selon les suppositions qui sont faites durant l'implémentation des différents outils d'analyse du code, les dépendances entre méthodes/classes capturées peuvent varier énormément [LLL08].

Ce problème de capture des dépendances impacte directement le calcul des métriques de couplage. En effet, ces métriques sont définies à partir de ces dépendances. Ainsi, selon les suppositions qui sont faites durant l'implémentation de ces métriques dans les différents outils et la façon dont on capture les dépendances entre méthodes/classes, les résultats peuvent varier anormalement d'un outil à un autre [LLL08].

1.2.2 Critères d'évaluation

La majorité des approches existantes ne se base que sur deux critères pour guider leur processus d'identification. Ces deux critères sont le couplage et la cohésion : les composants doivent être faiblement couplés entre eux et maximiser la cohésion entre leurs classes. Or, il existe d'autres critères pour évaluer la qualité d'un composant tel que sa non-appartenance à un cycle, sa cohésion sémantique, sa granularité, etc.

Malheureusement, utiliser une multitude de critères peut poser de nouveaux problèmes. Un de ces problèmes est le fait que certains critères sont contradictoires. Par exemple, un système sans cycles entre composants implique un système avec peu de composants contenant beaucoup de classes et donc des composants peu cohésifs. Au contraire, une forte cohésion tend vers beaucoup de petits composants et donc la multiplication des cycles. De plus, plus nous avons de critères pour évaluer une ABC, plus il est difficile de les agréger afin d'évaluer la qualité d'une solution.

1.2.3 Nature du système cible

Un autre problème qui peut survenir durant l'identification d'une ABC est lié à la nature du système cible. En effet, le paradigme OO peut être un problème en soi. Cela a été souligné par Lorenz : *"A good object-oriented design does not necessarily make a good component-based design, and vice versa"* [LV01]. Ainsi, les composants obtenus avec les approches se basant uniquement sur l'analyse statique restent conformes à la conception OO et représentent une vue de développement du système, tel que la répartition des classes dans les paquetages. Or l'objectif est d'obtenir des composants fonctionnels/logiques, c'est à dire, des composants correspondant à une abstraction de haut niveau sans tenir compte du design du système.

De plus, comme le fait remarquer Detten et al. [vDB11], le système cible n'est pas forcement parfait et peux contenir des *bad smell* qui peuvent fausser le processus d'identification.

1.2.4 Projection vers un modèle concret

Après avoir identifié les composants d'un système, il peut être intéressant de les projeter vers un modèle concret afin d'avoir une application opérationnelle à base de composants. A ma connaissance, aucune approche ne le propose. Seuls Washizaki et al. [WF05] permettent d'extraire certains composants de l'application pour les réutilisés dans de nouvelle application. Mais cette approche ne permet pas d'effectuer une réingénierie de l'application OO cible en application à base de composants.

Ainsi, cette étape reste à traiter, mais engendre de nouveaux problèmes. En effet, avant de projeter un composant vers un modèle concret, il faut identifier toutes les dépendances du composant, définir les interfaces requises et fournies, mettre en place un mécanisme de gestion de ces dépendances/interface et surement effectuer une réingénierie d'une partie du code (par exemple, les interfaces requises). Ainsi, un composant "concret" ne se résume pas simplement à un ensemble de classe identifié.

1.3 Apports de la thèse

Dans cette thèse, je propose un processus semi-automatique pour l'identification d'une architecture à base de composants. L'identification de cette ABC a pour but d'aider à la compréhension de l'application OO cible mais peut aussi servir de modèle pour une restructuration complète de l'application OO vers une application à base de composants. Pour chaque étape de l'identification d'une ABC (voir Figure 1.1) une ou plusieurs méthodes (étape 2 de la figure 1.1) sont proposées. Le processus complet se base sur des données obtenues avec une analyse dynamiques pour guider l'identification d'une ABC. Mais il utilise aussi des données statiques pour compléter cette identification.

Les ABC identifiées sont conformes à un modèle abstrait de composant. Ce modèle, bien que simple, contient toutes les notions importantes du paradigme composant. Ainsi, ce modèle aide la compréhension des ABC identifiées. De plus, la simplicité de ce modèle facilite la projection de l'ABC vers tout autre modèle de composants concret.

Pour l'étape d'identification des composants, je propose trois méthodes différentes. Comme nous le verrons dans le chapitre 6, les trois méthodes ont été développées successivement. Chaque nouvelle méthode vient résoudre des problèmes identifiés durant la validation de la méthode précédente. La première méthode d'identification des composants utilise un treillis de Galois et une heuristique spécialement développée afin de déterminer quel sont les composants de l'application cible. La seconde méthode utilise deux méta-heuristiques : un algorithme génétique suivi de l'algorithme *recuit simulé*. Ces deux méthodes utilisent le rapport cohésion/couplage pour identifier les composants. A ce niveau, l'originalité de mon approche réside dans l'utilisation de données

obtenues avec une analyse dynamiques. Enfin, la troisième méthode, la plus aboutie, s'appuie sur le principe de sélection multi-critères. En effet, avec cette méthode quatre critères sont utilisés pour évaluer une solution : le couplage, la cohésion , la granularité des composants et le nombre de cycles entre les composants. Contrairement aux méthodes précédentes, elle nous fournit un ensemble de solutions et non une solution unique. Dans cet ensemble, l'utilisateur de l'approche choisit la solution qui lui convient le mieux. cette approche utilise un algorithme génétique multi-objectifs basé sur le principe d'optimun de Pareto.

Pour guider le processus d'identification, j'ai fait le choix d'utiliser des données dynamiques. Ce choix repose sur la conviction que les analyses statiques ont du mal à capturer correctement les relations entre classes. Je mets en évidence le problème lié à l'analyse statique dans une étude de cas. Cette dernière est effectuée dans le cadre du calcul des métriques de couplage. Ces métriques sont utilisées, entre autres, dans des approches d'identification d'une ABC.

La dernière contribution de cette thèse porte sur l'utilisation de l'ABC obtenue pour effectuer une réingénierie de l'application cible en application orientée composant selon un modèle concret de composants.

1.4 Structuration du document

Après ce chapitre d'introduction présentant les motivation et la problématique lié à l'identification d'une ABC, le reste du document est organisé comme suit.

Le contexte général de cette thèse est présenté dans la partie II. Plus précisément, les définitions nécessaires à la bonne compréhension de ce contexte sont données dans le chapitre 2. Le chapitre 3 présente l'état de l'art des différentes approches liées à notre thématique. Cela regroupe l'identification de composants, l'identification d'une ABC ainsi que quelques approches de restructuration de logiciels non spécifiques au paradigme composant. De plus, les différents aspects de ces approches sont discutés afin de

mettre en évidence les problèmes encore non résolues.

La partie III présente en détail la contribution de ma thèse : un processus semi-automatique d'identification d'une architecture à base de composant dans une application orientée objet. Dans le chapitre 4 une présentation générale de ce processus est donnée. Le chapitre 5 présente en détail les méthodes et algorithmes utilisés pour obtenir les données qui guident ce processus. Le chapitre 6 présente successivement, et en détail, les trois méthodes d'identification des composants développés, l'étape de raffinement manuelle des composants et enfin l'étape d'identification des interfaces des composants.

Dans la partie IV, je présente les travaux réalisés afin de valider le processus d'identification d'une ABC, proposé dans ce document, ainsi que les différents choix faits durant le développement de ce processus. Le chapitre 7 présente une étude de cas qui a pour but la mise en évidence des problèmes liés au graphe d'appels construit statiquement. Cela est réalisé sur le calcul de la métrique de couplage CBO. Le chapitre 8 présente plusieurs études de cas qui ont pour but d'évaluer l'étape d'identification des composants présentée dans la section 6.3. Enfin, le chapitre 9 donne un exemple de faisabilité de restructuration de l'architecture à base de composant identifier vers un modèle concret de composant. Dans ce cas, le framework OSGi.

Enfin, dans la partie V, les principaux apports ce travail de thèse sont présentés, ainsi que ses limites et les perspectives d'améliorations et d'évolutions possibles.

Deuxième partie

Contexte

CHAPITRE 2

DÉFINITION ET HYPOTHÈSE

Dans ce chapitre, je présente le contexte général de cette thèse ainsi que les définitions nécessaires à sa bonne compréhension.

2.1 Identification d'une ABC dans un code orienté objets

Comme nous l'avons vu dans l'introduction, l'objectif de cette thèse est l'identification d'une architecture à base de composants dans une application orientée objet. Ceci est une tâche de rétro-ingénierie et plus précisément de rétro-conception. Chikofsky et al. [CCI90] donne la définition suivante de la rétro-ingénierie (Définition 1).

Définition 1. *La **rétro-ingénierie** (*reverse engineering*) est le processus d'analyse d'un système afin :*

- *d'identifier ses composants et leurs relations, et*
- *de créer une représentation de ce système sous une autre forme ou à un niveau d'abstraction plus élevé.*

La rétro-ingénierie implique généralement l'extraction d'artefacts de conception et la construction ou synthèse d'une abstraction moins dépendante de leurs implémentations. Elle peut être appliquée à n'importe quel niveau d'abstraction du système ou cycle de développement. La rétro-ingénierie est un processus d'exploration et non de modification. Elle a de nombreux sous-domaines, dont deux qui nous intéressent plus particulièrement : **la redocumentation** et la **rétro-conception**.

- **La redocumentation** est la création ou la révision d'une représentation sémantiquement équivalente au même niveau d'abstraction. La redocumentation est la forme la plus ancienne et la plus simple de la rétro-ingénierie.

- **La rétro-conception** (*design recovery*) identifie des abstractions pertinentes et de plus haut niveau du système. La rétro-conception peut reproduire toutes les informations requises pour parfaitement comprendre le comportement du système.

L'identification d'une ABC est une tâche de rétro-conception et non de redocumentation. En effet, le but d'une telle approche est de trouver une architecture à base de composants dans un système OO. Étant donné qu'une telle représentation n'existe pas dans la documentation, il s'agit d'une tâche de rétro-conception.

La projection de l'ABC d'une application objet vers un modèle concret de composant est un des objectifs secondaires de cette thèse. Ceci est une tâche de réingénierie. De nouveau, Chikofsky et al. [CCI90] nous en donnent une définition (Définition 2).

Définition 2. *La **réingénierie** (reengineering) concerne l'exploration et la modification d'un système cible pour le reconstituer dans une nouvelle forme ainsi que l'implémentation de cette nouvelle forme.*

La réingénierie inclue généralement un type de rétro-ingénierie (pour obtenir une représentation plus abstraite) suivie d'un type d'ingénierie directe (Définition 3) ou restructuration (Définition 4). Cela peut inclure des modifications pour respecter de nouveaux besoins non supportés par le système original.

Définition 3. *L'**ingénierie directe** (forward engineering) est le processus traditionnel qui consiste à passer d'abstractions de haut niveau, au design indépendant de l'implémentation, vers l'implémentation physique d'un système.*

Définition 4. *La **restructuration** (restructuring) est la transformation d'une représentation vers un autre niveau d'abstraction relativement égal, tout en préservant le comportement externe du système cible.*

La réingénierie d'une application OO vers une application à base de composants consiste en une tâche de rétro-conception (identification de l'ABC de l'application cible) puis d'une tâche d'ingénierie directe (projection de l'ABC de haut niveau d'abstraction dans un niveau plus bas : un modèle de composants concrets).

2.2 Architecture logicielle

La définition la plus utilisée pour une architecture est celle de Bass et al. [BCK03] (Définition 5).

Définition 5. *L'**architecture logicielle** d'un programme ou d'un système est la ou les structures du système, c'est-à-dire les éléments logiciels, les propriétés visibles extérieurement de ces éléments et leurs relations.*

Celle-ci, très abstraire, décrit l'architecture d'un système à l'aide d'éléments logiciel qui la constituent et les relations entre eux. Ces éléments logiciels, leurs propriétés ainsi que leurs relations ne sont pas définis plus en détail.

La Définition 6 de Perry et al. [WFP+92], bien que plus ancienne, décrit plus clairement les éléments qui composent une architecture.

Définition 6. *L'**architecture logicielle** est un ensemble d'éléments architecturaux (ou, si vous préférez, de conception) qui ont une forme particulière. Nous distinguons trois classes différentes d'éléments architecturaux :*

- *les éléments de calcul ;*
- *les éléments de données ; et*
- *les éléments de connexion.*

La définition 7 donne une définition communément admise d'une architecture.

Définition 7. *L'**architecture** est une vue abstraite d'un système en terme d'éléments architecturaux. Ces éléments sont :*

- *les composants qui décrivent les fonctionnalités métier de l'application ;*
- *les interface qui décrivent les communications et connexions entre les composants ;*

- *la configuration qui décrit la topologie des connexions entre composants et connecteurs.*

Par la suite, quand je parlerai d'architecture, je me référerais à la définition 7. Les principaux éléments que je manipule y sont décrits. De plus, cette définition reste suffisamment abstraite pour être adaptée aux différentes approches d'identification d'architecture rencontrées.

Une architecture logicielle joue un rôle important dans différents aspects du développement d'une application [Gar00]. Ceux-ci sont : **la compréhension**, **l'analyse**, **la construction**, **la réutilisation**, **l'évolution** et enfin **la gestion de projet**. Dans le cadre cette thèse, trois de ces aspects n'intéressent plus particulièrement :

- **La compréhension** : l'architecture logicielle permet une compréhension plus aisée du système cible. En effet, elle représente les systèmes dans un niveau l'abstraction élevée où les fonctionnalités de haut niveau du système ainsi que leurs relations peuvent être facilement comprises. Ceci facilite considérablement la compréhension de l'organisation de systèmes assez larges et complexes.

- **L'évolution** : l'architecture fournit un squelette du système et expose les possibilités d'évolution du système. Ceci permet de mieux gérer la propagation des changements et d'évaluer les coûts associés à l'évolution. Ainsi, l'architecture permet de mettre en valeur les parties nécessitant une attention particulière lors de l'évolution du système.

- **La réutilisation** : une architecture logicielle permet également de déterminer quel sont les composants réutilisables et comment les réutiliser. En effet, elle met en évidence les dépendances et les services fournis par les composants à travers leurs interfaces.

2.3 Composant logiciel

Une première définition des composants (Définition 8) est donnée par Szyperski [Szy02]. Celle-ci se focalise sur la composition, les interfaces contractuelles et le déploiement des composants. On se place dans un contexte d'assemblage où ces entités (composants) ont déjà été développées et où l'on voudrait les assembler pour former une application. Les composants spécifient des interfaces sous forme de contrats, et peuvent être déployés séparément.

Définition 8. *Un **composant** est une unité de composition qui spécifie par contractualisation ses interfaces, et qui explicite ses dépendances de contexte uniquement. Un composant logiciel peut être déployé indépendamment et est sujet à une composition par des tierces entités.*

Une autre définition est celle de la spécification UML 2.0 [OMG05] (Définition 9).

Définition 9. *un **composant** est une partie modulaire d'un système qui encapsule son contenu et dont la manifestation est remplaçable dans son environnement. Un composant définit son comportement en terme d'interfaces fournies et requises. En tant que tel, un composant sert comme un type, dont la conformité est définie par ses interfaces fournies et requises (incluant à la fois leur sémantique statique et dynamique)*

Celle-ci met l'accent sur la modularité, la substitution et le typage. Ici, on se place dans un contexte de modélisation architecturale ou l'on perçoit le système comme étant un ensemble d'unités modulaires et substituables. Chaque unité (composant) est typée par ses interfaces, qui spécifient ces services requis et fournis.

La définition 10 donne une définition communément admise d'un composant.

Définition 10. *Un **composant** est une entité logicielle qui fournit une fonctionnalité de haut niveau. Il définit son comportement et ses besoins à travers des interfaces requises et fournies.*

Par la suite, quand je parlerais de composant j'utiliserais la définition 10. Celle-ci est cohérente avec la définition d'une architecture que j'utilise (Définition. 7). Cette définition est suffisamment abstraite pour être utilisée dans l'identification d'une ABC tout en définissant les concepts clés d'un composant. De plus, cette définition est compatible avec les spécifications des modèles de composants concrets.

2.3.1 Interface

Contrairement au composant, il n'existe pas de définition communément admise des interfaces. Par exemple, dans CCM [Gro06], où l'on parle de facettes et de réceptacles alors que dans Fractal [BCS04] on parle d'interfaces client et serveur. La définition 11 donne une définition simple d'une interface.

Définition 11. *Les **interfaces** sont la partie visible d'un composant. Elles servent à déclarer les services fournis et requis par le composant.*

J'utiliserai par la suite la Définition 11 quand je parlerai d'interface.

Ainsi, les interfaces définissent le comportement et les besoins d'un composant. Les interfaces requises définissent les fonctionnalités (service) que l'on doit obligatoirement lui fournir pour fonctionner correctement. Au contraire, les interfaces fournies décrivent les fonctionnalités (service) que ce composant peut fournir. Une interface est composée d'un ou plusieurs services.

Par exemple, dans la Figure 2.1 le composant $Lexer$ possède deux interfaces fournies. L'interface fournie $Token$ fournit deux services $getCurrentToken()$ et $getNextToken()$, ces deux services fournissent deux fonctionnalités liées à la gestion des tokens. Le composant a une interface requise $CharStream$ dont le service $setCharStream()$ permet d'initialiser le flux de caractères nécessaires à la génération des tokens.

2.3.2 Connecteur

Comme les composants ou les interfaces, les connecteurs sont un élément de premier plan d'une architecture. En effet, alors que les interfaces définissent les services

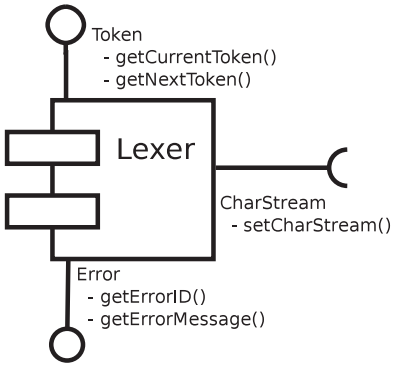


Figure 2.1 – Exemple de composants selon la définition 10

requis et fournis d'un composant, les connecteurs définissent les communications entre les composants. Shaw et al. [SG96] nous en donnent une définition (Définition. 12).

Définition 12. *Les **connecteurs** gèrent les interactions entre les composants. Ils établissent les règles qui gouvernent les interactions entre composants et spécifient tous les mécanismes auxiliaires nécessaires.*

Ainsi, un connecteur implémente la sémantique de l'interaction entre les composants et couvre les besoins d'adaptation entre les composants. Les connecteurs peuvent modéliser un simple appel de méthode entre deux composants ou être plus complexes. Par exemple dans Fractal [BCS04], un connecteur peut adapter une interface fournie à une interface requise dans le cas ou leurs services n'ont pas exactement la même sémantique. De plus, il faut noter que certains modèles (par exemple EJB) ne définissent par formellement la notion de connecteur.

2.4 Modèle de composants utilisé

Par la suite, j'utiliserai le modèle de composants de la figure 2.2. Ce modèle est conforme aux définitions d'architecture (Définition. 7) et de composant (Définition. 10) que j'ai décidé d'utiliser et contient ainsi les principales notions du paradigme composant. Les connecteurs ne sont pas définis dans ce modèle. Comme nous venons de le

voir, les connecteurs gèrent les communications entre les composants. Dans mon cas, un connecteur modéliserait soit un appel de méthode, soit un accès à un attribut. Modéliser cela n'apporte aucune nouvelle information pertinente.

De plus, ce modèle est extrêmement simple ce qui permet de le projeter très facilement vers tout autre modèle de composants.

2.5 Données du problème

Ci-dessous sont données les définitions de trois types de donnée couramment utilisés dans l'identification d'une ABC.

Définition 13. *Un **cas d'utilisation** décrit une fonctionnalité fournie par un système qui donne un résultat visible pour un acteur. Une instance d'un cas d'utilisation capture une séquencer spécifique d'interaction, cette instance est appelée scénario*

Définition 14. *Une **trace d'exécution** est un arbre orienté $T(V, E)$ où les noeuds V représentent les exécutions des méthodes et les arcs $E \in V \times V$ les appels entre ces méthodes. Elle correspond à l'arbre des appels entre méthodes associées à une exécution donnée de l'application.*

Définition 15. *Un **graphe d'appels** d'un système est un graphe orienté $G(V, E)$ où les noeuds V représentent les méthodes du système et les arc $E \in V \times V$ les appels entre ces méthodes.*

Définition 16. *L'**analyse statique** d'un système couvre une variété de procédés utilisées pour obtenir des informations sur un programme sans l'exécuter.*

Le graphe d'appels d'une application peut être obtenu avec une analyse statique, dans ce cas on parle de graphe d'appels statique ou à partir d'une trace d'exécution, dans ce cas on parle de graphe d'appels dynamique.

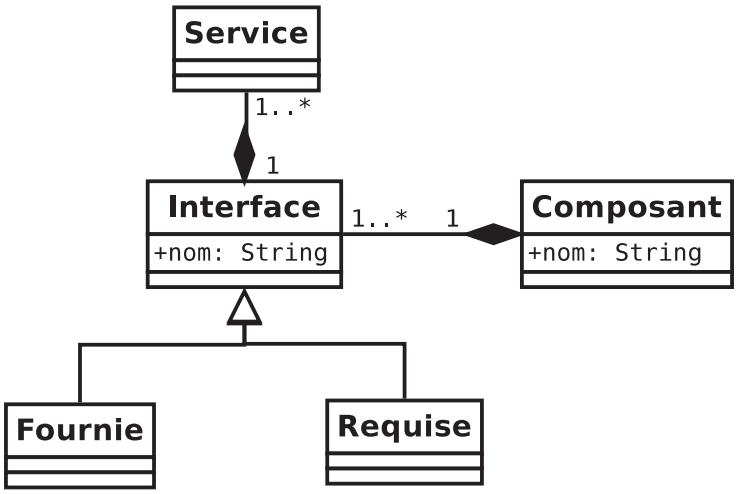


Figure 2.2 – Modèle de composants utilisé

CHAPITRE 3

ETAT DE L'ART

Dans ce chapitre je présente les différentes approches liées à la thématique de cette thèse en fonction de 5 critères. Cela regroupe l'identification de composants, l'identification d'une ABC ainsi que quelques approches de restructuration de logiciels non spécifiques au paradigme composant. Ensuite les différents aspects de ces approches seront discutés afin de mettre en évidence les problématiques restantes.

3.1 Principes communs

Toutes les approches d'identification de composants ou d'architectures à base de composants utilisent la même base pour leur définition d'un composant dans un système OO : *un composant est un ensemble de classe fournissant une fonctionnalité*.

Ensuite, cette définition peut être complétée. Comme nous allons le voir, certaines approches vont plus loin et définissent les interfaces requises et fournies d'un composant : les interfaces requises des composants sont les méthodes/attributs déclarées dans des classes qui n'appartiennent pas au composant, mais qui sont utilisées par des classes du composant. De même, les interfaces fournies correspondent aux méthodes/attributs des classes du composant utilisées par des classes extérieures au composant. Dans ce cas, un connecteur correspond à un appel de méthode ou un accès à un attribut. Ils sont donc implicites et ne sont pas modélisés.

Par exemple, la Figure 3.1 représente les composants d'une application OO. Les noeuds correspondent aux classes, les arcs correspondent aux dépendances entre classes. Cette application est constituée de trois composants. Le composant $comp1$ est formé par les classes A, Z et E. Les classes Z et E font appel à deux classes extérieures : I et R. Ainsi, ce composant a deux interfaces requises, fournie par l'interface fournie R du composant $comp2$ et l'interface fournie I du composant $comp3$.

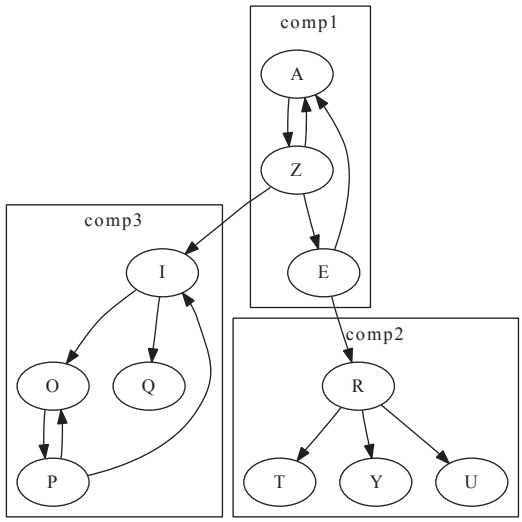


Figure 3.1 – Composant dans une application OO

Toutes les approches utilisent le même schéma général d'identification des composants : les relations et dépendances entre les classes sont obtenues à partir des différents artefacts du système cible. Même si cela n'est pas toujours présenté comme tel, cet ensemble de classes/relations peut être vu comme un graphe où les noeuds représentent les classes et les arcs les relations de dépendance. Ensuite, des heuristiques ou des algorithmes sont appliqués à ce graphe afin d'obtenir une partition de l'ensemble des classes. Chaque sous-ensemble de la partition est un des composants identifiés. L'identification de cette partition est guidée par différents critères tels qu'une forte cohésion dans les composants ou un couplage faible entre composants. Enfin, ces ensembles de classes identifiées peuvent être projetés dans une architecture orientée composante ou même un modèle de composants concret. Cela est fait en identifiant les interfaces fournies et requises des composants. Ainsi, l'identification d'une ABC peut-être décomposée en trois grandes étapes (voir Figure 1.1) :

1. Construction du graphe de relations à partir des différents artefacts du système cible.

2. Identification des composants à l'aide d'algorithmes/heuristiques.

3. Identification de l'architecture formée par les composants identifiées.

Plutôt que de présenter les différentes approches d'identification d'une architecture à base de composant selon les trois étapes du schéma global, j'ai préféré les présenter selon cinq critères différents. Je considère que ces cinq critères permettent de mieux appréhender les différences entre les approches. En effet, ces cinq critères couvrent les trois étapes du processus d'identification d'une ABC tout en mettant en évidence certains points qui ne sont pas toujours bien explicités (par exemple, l'utilisation ou non d'un modèle de composants). Ces cinq critères sont :

- **Complétude de l'analyse** : l'identification des composants porte-t-elle sur toute l'application ou uniquement sur une sous-partie ?
- **Degré d'automatisation** : le processus d'identification est-il automatique ou semi-automatique ?
- **Modèle de composant cible** : quel modèle l'approche utilise-t-elle réellement ?
- **Artefact utilisé pour l'identification** : quels sont les artefacts (données) qui guident le processus d'identification ?
- **Méthode d'identification des composants/de l'architecture** : quels algorithme ou heuristique est utilisé pour l'identification ?

Comme toutes les approches n'ont pas forcément de nom, je me réfère à elles en utilisant le nom des auteurs et/ou références du papier qui les présente.

3.2 Complétude de l'analyse

Les approches d'identification effectuent soit une analyse globale du système, soit une analyse locale. En effet, certaines approches [JCIR01, JKL03, KC04, FCDCXF05, MJ06, CSTO08, CKK08, vDB11] ont comme objectif d'identifier tous les composants du système. Ainsi, ils partitionnent l'ensemble des classes du système en vue d'obtenir

une cartographie globale du système OO dans le paradigme composant. Ces approches sont similaires à celles qui ciblent le clustering de logiciel [MB07, MM06, PHLR09], mais avec l'ajout du concept de composant. D'un autre côté, certaines approches n'ont comme objectif que d'identifier un sous-ensemble des composants du système. Dans ce cas, ils s'appuient sur une analyse locale, en se focalisant sur une partie ou une certaine fonctionnalité du système. Par exemple, l'approche proposée par Lee et al. [LSK+01] permet l'identification des composants liés à un package donné. La délimitation du sous-système cible peut être aussi réalisée par un élément de plus faible granularité. Par exemple, l'approche proposée par Washizaki et al. [WF05] identifie le composant qui fournit une fonctionnalité précise. Cette fonctionnalité est déterminée à partir de la classe considérée comme point d'entrée de celle-ci. Une telle approche est semblable aux approches de localisation de fonctionnalité [GD05, PGM+07, LMPR07, EKS03] dont le but est d'identifier les classes, méthodes, attributs, etc. impliqués dans une fonctionnalité précise. La principale différence est l'introduction du concept de composant.

3.3 Degré d'automatisation

Selon le processus d'identification, l'intervention d'un expert peut être ou non nécessaire. Un expert peut être toute personne ayant une connaissance approfondie du système. Ainsi, les approches peuvent être classées en deux catégories : automatique ou semi-automatique.

Une approche est considérée comme automatique si aucun expert n'est requis durant le processus d'identification ou si l'intervention de celui-ci n'est que facultative. Par exemple, l'outil Bunch [MM06], propose automatiquement une décomposition d'une application sous la forme de module et cela à partir de son code source. Dans le cas de l'approche proposée dans [CSTO08], une proposition d'ABC est faite automatiquement à partir du graphe de relation entre classes. Ensuite, cette ABC peut être raffinée par un expert à l'aide de la documentation du système cible. Cette étape étant facultative.

Dans le cas des approches semi-automatiques, un expert doit obligatoirement intervenir à un moment donné du processus d'identification. Par exemple, dans l'approche

proposée par Patel et al. [PHLR09], un expert du système doit définir les principaux cas d'utilisation afin de générer les traces d'exécution. Dans les approches proposées par Jain et al. [JCIR01] et Medvidovic et al. [MJ06], l'identification de ABC est basée sur un ensemble de règles prédéfinies. Le choix des règles à appliquer est fait par un expert du système. Dans les approches [CKK08, vDB11] le regroupement des classes sous la forme de composant se fait de façon incrémentale et itérative : à chaque itération, les nouveaux composants formés doivent être validés par un expert. De même, les approches proposées dans [LSK+01, KC04, WF05, FCDCXF05] sont considérées comme semi-automatiques car elles proposent un ensemble de composants candidats et l'expert doit choisir les composants finaux dans cet ensemble.

3.4 Modèle de composants cible

Au début de cette section, il a été dit que toutes les approches définissent un composant comme un ensemble de classes et ces classes sont regroupées en fonction de leurs dépendances. C'est une définition très générique d'un composant. Conformément aux différentes définitions d'un composant et d'architecture à base de composant donnée dans la section 2, cela ne suffit pas pour former une ABC. En effet, l'information sur les interfaces requises et fournies, qui permet la composition des composants et ainsi la construction d'une architecture, est manquante.

En fonction de leur objectif final, les différentes approches d'identification de composant utilisent soit la définition générique d'un composant dans une application OO ou un modèle plus complet. Ainsi, les approches [LSK+01, JCIR01, KC04, FCDCXF05] utilisent le modèle générique étant donné que leur principal objectif est l'aide à la compréhension du système. Comme les approches [MM06, PHLR09] sont des approches de décomposition des applications sous la forme de modules, elles ne possèdent pas la notion de composant. Mais la définition qu'elle donne des modules est similaire à la définition générique d'un composant dans une application OO. Ainsi, ces deux approches utilisent aussi la définition générique.

Quand un modèle de composants plus précis est utilisé, l'objectif est souvent l'exploitation des propriétés du modèle (par exemple structurelle) afin d'améliorer la phase d'identification. Par exemple, l'approche proposée dans [WF05] projette les ensembles de classe identifiée vers le modèle de composant JavaBean. Ainsi, cette approche doit définir la notion de dépendance entre les classes de façon plus précise (par exemple, différentier un appel de méthode par rapport à un accès d'attribut). Les approches [MJ06, CSTO08] sont basées la définition générique d'un composant dans une application OO et définissent ne plus la notion d'interfaces. Ainsi, leur modèle est à mi-chemin entre le modèle générique "basique" et un modèle concret.

Certaines approches [JKL03, MJ06] définissent plusieurs types de composants. Par exemple, Jang et al. [JKL03] distingue deux types de composants possible : les composants données et les composants métiers.

Enfin, l'approche [CKK08] et son extension [vDB11] utilisent un modèle basé sur le modèle de composant Palladio [RBK+07]. Ainsi, leur modèle définit clairement le mode de communication entres composants qui se fait à travers des interfaces qui sont elles-mêmes constituées de services. De plus, ce modèle permet aux composants d'être composite (un composant peut être constitué de plusieurs sous composants). Toutes ces notions sont bien-sûr utilisées durant la phase d'identification de l'ABC.

3.5 Artefacts utilisés pour l'identification

Les relations et les dépendances entre les classes, qui permettent l'identification des composants, peuvent être obtenues à partir de trois types d'artefacts logiciels différents : la documentation, les traces d'exécution et le code source.

3.5.1 Documentation

Dans certaines approches, la documentation est considérée comme la principale source pour l'extraction de relations entre les classes. Celle-ci inclut les diagrammes de classes, les diagrammes de séquences, les *domain business models* et les cas d'utilisation. Par exemple, Jain et al. [JCIR01] identifient des composants métier à partir des relations

entre classes dans le domain business model de l'application cible.

En plus des relations entre les classes, la documentation peut fournir des relations de haut niveau entre les fonctionnalités (plus abstrait que les appels de méthodes entre classes ou les liens d'héritage) du système. Ces caractéristiques sont ensuite reliées aux classes qui les implémentent. Par exemple, dans l'approche de Kim et al. [KC04] les cas d'utilisation sont utilisés pour mesurer les dépendances entre les différentes fonctionnalités et ensuite les regrouper. Ensuite, les diagrammes de séquences sont utilisés pour assigner les classes à la fonctionnalité qu'elles fournissent.

Enfin, la documentation textuelle peut être utilisée pour raffiner manuellement les composants identifiés [CSTO08], constituant ainsi une étape de post-traitement.

3.5.2 Traces d'exécution

Ces artefacts sont obtenus en capturant les appels entre les méthodes, les initialisations objet, les lectures/écritures sur les attributs, etc. durant l'exécution du système cible. Les traces d'exécution fournissent les relations dynamiques entre les classes. Ces traces sont obtenues soit en exécutant des cas d'utilisation, soit durant une utilisation réelle du système par un utilisateur. Cependant, pour des raisons pratiques, la majorité des approches obtient les traces d'exécution à partir des cas d'utilisation. Dans ce cas, les cas d'utilisation sont soit tirés de la documentation, soit proposés par un expert [PHLR09]. De plus, Patel et al. [PHLR09] associent chaque trace d'exécution capturée à son cas d'utilisation. Ainsi, toutes les classes et relations contenues dans la trace sont associées aux fonctionnalités mises en oeuvre par le cas utilisation.

3.5.3 Code Source

À partir du code source, il est possible d'extraire différents types de relations de bas niveau entre les classes. Ces relations peuvent être les appels de méthodes, l'arbre d'héritage, les lectures/écritures sur les attributs, ou bien encore les agrégations. Par exemple, dans les approches présentées dans [WF05, CSTO08, CKK08, vDB11], le graphe représentant les relations entre les classes est produit à partir d'une analyse du

code source. Les approches [CKK08, vDB11] prennent aussi en compte le vocabulaire utilisé dans les classes afin de déterminer leur similitude. Ces approches exploitent aussi le type des paramètres et le retour des appels de méthode entre classes.

3.5.4 Hybride

Parfois, pour construire le graphe de dépendance entre classes, différentes sources de données sont utilisées. C'est le cas de l'approche proposée par Lee et al. [LSK+01]. Dans ce cas, l'information obtenue à partir du code source est combinée avec des informations provenant des diagrammes de cas utilisation pour construire le graphe de dépendance. De même, Patel et al. [PHLR09] utilisent les relations statiques entre les classes afin de compléter les modules identifiés à l'aide de données dynamiques.

3.6 Méthodes d'identification

Comme nous l'avons vu précédemment, l'identification des composants est basée sur la partition des classes de l'application cible dans le but d'obtenir des ensembles (composants) fortement cohésifs et peu couplés avec les autres ensembles. Or, trouver la partition optimale est un problème NP-difficile [GJ79]. En effet, le nombre de partitions possible est exponentiel en fonction du nombre de classes. Soit n le nombre d'éléments à partitionner et k le nombre maximal de sous-ensembles, le nombre de partitions possible est donné par le nombre de Stirling de seconde espèce [NW78] :

$$G_{n,k} = \begin{cases} 1 & si\ k = 1\ ou\ k = n \\ G_{n-1,k-1} + kG_{n-1,k} & sinon \end{cases} \tag{3.1}$$

Par exemple, pour 5 classes il y a 52 partitions possible, pour 15 classes 1 382 958 545 partitions possible. Ainsi, pour tout problème de taille réaliste, il est clairement impossible d'utiliser une méthode exhaustive.

Une première solution pour effectuer ce partitionnement est d'appliquer un algorithme de partitionnement (hiérarchique [JCIR01] ou non [LSK+01]) sur le graphe des

relations entres classes. L'algorithme utilisé peut être spécialement développé pour le clustering logiciel, et donc prend en compte les contraintes spécifiques liées à ce domaine [LSK+01] ou être un algorithme généraliste [FCDCXF05], utilisé dans d'autres domaines.

Parmi les différentes approches, beaucoup utilisent des méthodes ou algorithmes non spécifiquement développés pour le partitionnement. Parmi celles-ci, certaines sont basées sur des heuristiques de recherche. Cette famille d'algorithme inclut des algorithmes stochastiques itératifs. Ceux-ci progressent vers un optimum global par échantillonnage d'une fonction objectif. La fonction objectif évaluant la qualité d'une solution. Dans ce cas, la difficulté réside dans la définition de la fonction objectif. En effet, il guide l'ensemble du processus. Par exemple, Chardigny et al. [CSTO08] utilisent l'algorithme du recuit simulé [MRR+53]. La fonction objectif de cette approche utilise des critères de qualité telle que la composabilité, maintenabilité, fiabilité, etc. Ces caractéristiques sont évaluées par des métriques existantes qui capturent la complexité, la cohésion, etc.

De même, dans l'outil Bunch l'ensemble des classes peut être partitionnée en utilisant différentes méta-heuristiques (algorithme hill-climbing [MRR+53] ou génétique [Hol75]). La fonction objectif utilisée est basée sur le rapport cohésion/couplage.

D'autres approches utilisent des techniques de fouille de donnée dans le but de trouver des ensembles de classe qui partagent les mêmes propriétés. Par exemple, Patel et al. [PHLR09] recherchent des ensembles de classes participant à la même fonctionnalité (en pratique, au même cas d'utilisation).

Certaines approches utilisent des heuristiques spécifiquement développées pour l'identification des composants. Par exemple, dans les approches [JCIR01, MJ06], l'identification est faite à partir d'un ensemble de règles, ces règles définissant la façon de partitionner les classes du système. Elles groupent les classes en fonction de leur type de relation. Par exemple dans [MJ06], si deux classes ont une relation d'héritage, elles sont regroupées dans le même ensemble.

Dans [CKK08], Chouambe et al. proposent un processus itératif qui regroupe deux à deux les composants obtenus à l'étape précédente (les premiers composants étant les couples classe/interface). Le regroupement se fait à l'aide d'une métrique de similarité entre composants qui prend en compte : la cohésion, la métrique *distance from main sequence* [Mar], appartenance au même package des classes qui le compose, similitude du nom des classes/interfaces du composant, etc. Dans [WF05], Washizaki et al. proposent un algorithme qui, à partir des relations entre les classes et une fonctionnalité voulue, donne un ensemble de composants candidats.

Enfin, il faut noter que certaines approches utilisent des combinaisons des techniques présentées ci-dessus pour identifier les composants. Par exemple, Jain et al. [JCIR01] utilisent un algorithme de partition hiérarchique, algorithme non spécialisé dans l'identification de modules/composants dans une application, suivi d'un ensemble de règles à appliquer sur les composants identifiés pour les raffiner. Dans [PHLR09], Patel et al. utilisent un algorithme de fouille de données sur des traces d'exécution, suivie d'un algorithme qui assigne chaque classe non présente dans les traces (ces classes ne sont pas prises en compte dans l'étape précédente) à un des ensembles de classe identifiée précédemment [TH00]. Detten et al. [vDB11] étendent l'approche [CKK08] en ajoutant une phase de détection des mauvais patrons de conceptions entre les composants identifiés : si un *bad smell* est trouvé entre deux composants (plus précisément entre les classes de ces deux composants), l'application est restructurée et l'identification de composant est relancée.

Le tableau 3.I donne une synthèse des approches présentées dans cet état de l'art.

	Analyse	Auto	Modèle	Artefacts	Méthode
[LSK+01]	locale	non	définition générique d'un composant	diagramme des cas d'utilisations, diagramme de classe	heuristique de partitionnement
[JCIR01]	globale	non	définition générique d'un composant	diagramme des cas d'utilisations, diagramme de classe, diagramme de séquences	algorithme de partitionnement hiérarchique, heuristique (règles)
[JKL03]	globale	oui	définition générique d'un composant	cas d'utilisation, diagramme de classe	heuristique de partitionnement
[KC04]	globale	non	définition générique d'un composant	diagramme des cas d'utilisations, diagramme de séquences	analyse de matrice
[FCDCXF05]	locale	non	définition générique d'un composant	Domain busines model	algorithme de partitionnement hiérarchique
[WF05]	locale	non	modèle concret : JavaBean	graphe d'appels statiques, héritage, lecture/ecriture des attributs	heuristique de partitionnement
[MM06]	globale	oui	non basé sur un modèle de composants	graphe d'appels statiques	méta-heuristique : algorithme génétique, recuit simulée
[MJ06]	globale	non	modèle d'architecture à base de composants générique	diagramme de classe, graphe d'appels statiques	heuristique (règles)
[CSTO08]	globale	oui	modèle d'architecture à base de composants générique	graphe d'appels statiques, héritage, documentation	méta-heuristique : recuit simulée
[PHLR09]	global	non	non basé sur un modèle de composants	traces d'exécution obtenue à partir de cas d'utilisation proposé par expert, diagramme de classe	fouille de données, heuristique (algorithme orphan)
[CKK08]	globale	non	modèle de composant basée sur Palladio	graphe d'appels statiques, héritage, package, nom des classes	heuristique itérative de regroupement des classes/composants
[vDB11]	globale	non	modèle de composant basée sur Palladio	graphe d'appels statiques, héritage, package, nom des classes, *bad smell*	heuristique itérative de regroupement des classes/composants, restructuration de l'application

Tableau 3.I – Synthèse des approches en fonction des critères d'évaluation

3.7 Verrous identifiés

Après avoir présenté l'état de l'art, les problèmes déjà présentés dans l'introduction sont discutés plus en détail dans cette section.

3.7.1 Obtention des relations entre classes

Dans toutes les approches, l'obtention des relations entre classes depuis les différents types artefacts est considérée comme triviale. Or, en fonction des artefacts, de nombreux problèmes peuvent se poser.

3.7.1.1 Documentation

Les approches [LSK+01, JCIR01, KC04, FCDCXF05, CSTO08] basent leur processus d'identification totalement ou en partie sur la documentation. Or, comme le fait remarquer Medvidovic dans [MJ06], la documentation d'un système n'est pas toujours fiable. De plus, une enquête sur des logiciels industriels réalisée en 2002 a révélé que la satisfaction à l'égard de la qualité des documents est faible, voire très faible dans 84% des projets étudiés [VC02]. De même, de nombreuses études pointent la qualité et complétude de la documentation comme un problème majeur dans l'industrie, plus précisément, durant la phase de maintenance d'une application [LS, Cha85, Sou98, VC02]. Ainsi, utiliser la documentation comme donnée en entrée pour identifier une ABC me semble être une erreur. En outre, si le but est l'aide à la compréhension de l'application, cela est contre-productif. En effet, si la documentation est complète, mais non à jour, l'ABC obtenue représentera une version ultérieure de cette application. Si la documentation n'est pas complétée, l'ABC obtenu risque d'être fausse et non complète.

3.7.1.2 Graphe d'appels statique

Les appels de méthodes, utilisés dans de nombreuses approches [LSK+01, WF05, MJ06, CSTO08, PHLR09, CKK08, vDB11], sont le plus souvent capturés statiquement (par analyse du code), sous la forme d'un graphe d'appels. En effet, les graphes d'appels

statiques ont le grand avantage de couvrir toutes les classes du système et semble facile à obtenir. Or, les mécanismes dynamiques des langages objets, tel que le polymorphisme ou le chargement dynamique de classe posent de réels problèmes pour l'extraction exacte et complète des appels entre méthodes. Ainsi, un graphe d'appels statique peut contenir des appels de méthodes qui, en réalité, n'auront jamais lieux ou au contraire ne pas contenir des appels de méthode qui ont bien lieux.

Par exemple, les quatre graphes d'appel de la Figure 3.3 correspondent au code de la Figure 3.2. Ce code contient des appels polymorphes (`b.foo()` dans la méthode `Main.main`) et des instanciations dynamiques de classe (`c.newInstance()` dans la méthode `A.no(String)`). Comme l'on peut le voir, seul le graphe d'appels obtenu dynamiquement (Fig. 3.3a) est complet (il ne manque aucun appel) et correct (tous les appels ont réellement lieux). Les deux autres graphes d'appels sont calculés avec différents algorithmes d'analyse de type (ces algorithmes effectuent une analyse statique de l'application cible) : *Class Hierarchy Analysis* (CHA) [DGC95], un algorithme basique d'analyse de type et *Variable Type Analysis* (VTA) [SHR+00] un algorithme plus précis. Mais qui ne prend pas en compte l'instanciation dynamique. Selon l'algorithme utilisé, le graphe d'appels est soit incomplet (Fig. 3.3c), soit incorrect (Fig. 3.3b).

En effet, le graphe d'appels de la Figure 3.3b, construit avec l'algorithme CHA contient beaucoup trop d'appels. De plus, le graphe d'appels construit avec VTA est incomplet car cet algorithme ne prend pas en compte l'instanciation dynamique de l'objet de type `C` dans la méthode `A.no` et manque ainsi l'appel vers la méthode `B.bar` depuis la méthode `C.foo`.

Ainsi, utiliser un graphe d'appel statique pour identifier des ABC peut engendrer une explosion du nombre de dépendances entre classes ou au contraire, un manque de certaines de ces dépendances.

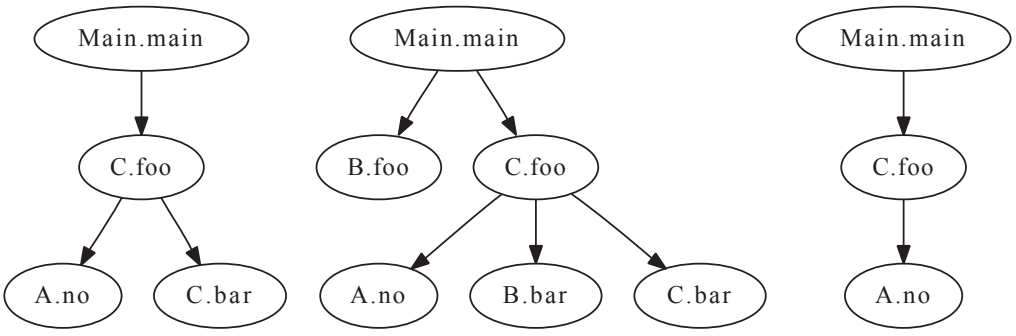
3.7.1.3 Traces d'exécution

Celles-ci posent des problèmes inverses aux données obtenues statiquement. En effet, même si l'on est certain qu'une relation obtenue à partir des traces d'exécution existe

```
class Main {
   void main(String[] args){
    B b = new C();
    b.foo();
  }
}

class A {
  Object no(String  cl){
    Class c = Class.forName(cl);
    return c.newInstance();
    }
}
```

```
class B {
  void foo(){}
  void bar(){}
}

class C extends B {

  void foo(){
    A a = new A();
    Object o = a.no("C");
    (B)o.bar();
  }
  void bar(){}
}
```

Figure 3.2 – Graphes d'appels de l'exemple de la Figure 5.1



(a) Dynamique (b) CHA (c) VTA

Figure 3.3 – Graphes d'appels de l'exemple de la Figure 3.2

réellement, il est impossible de savoir si toutes les relations entre les classes ont été obtenues. En effet, il est impossible de garantir la couverture totale de l'application cible. C'est pourquoi les approches qui les utilisent préconisent de les capturer en exécutant les cas d'utilisation [PHLR09]. Une solution possible est de compléter les données obtenues dynamiquement avec des données obtenues statiquement [PHLR09]. De plus, exécuter tous les cas d'utilisation de l'application peut-être longs est fastidieux, cela peut expliquer pourquoi très peu d'approches utilisent ce type de données.

3.7.2 Nature du système cible

Un autre problème qui peut survenir durant l'identification d'une ABC est lié à la nature du système cible. En effet, le paradigme OO peut être un problème en soi. Cela a été souligné par Lorenz : *"A good object-oriented design does not necessarily make a good component-based design, and vice versa"* [LV01] . Certains patrons de conceptions, comme adaptateur [GHJV95], peuvent aider à l'identification des composants. Dans ce cas, un tel patron de conception peut être vu comme un connecteur et définir ainsi la limite entre les deux composants qui l'utilisent. Au contraire, le patron de conception visiteur [GHJV95] peut augmenter la difficulté de l'identification des composants. Dans ce cas, la structure de donnée est fortement couplée à toutes les classes qui l'utilisent. Si la structure de données est utilisée par deux composants différents, ce patron de conception empêche l'identification de ces deux composants. Ainsi, même si un système est très bien implémenté selon l'approche OO, il peut causer des problèmes pendant le processus d'identification d'une ABC.

De plus, comme le fait remarquer Detten et al. [vDB11], le système cible n'est pas forcement parfait et peux contenir des *bad smells*. Ces *bad smells* peuvent gêner le processus d'identification. Par exemple, le *bad smell large class* consiste en une classe trop grande qui participe à plusieurs fonctionnalités. Une telle classe peut forcer deux composants distincts à fusionner durant le processus d'identification.

3.7.3 Critères d'évaluation d'une ABC

Comme nous l'avons vu dans la Section 3.1, les approches d'identification de composants ou de CBA existantes ne se basent généralement que sur deux critères pour guider leur processus. Ces deux critères sont le couplage et la cohésion : les composants doivent être faiblement couplés entre eux et maximiser la cohésion entre leurs classes. Or, il existe d'autres critères pour évaluer la qualité d'un composant et de l'architecture qui le contient. Ces critères peuvent être la non-appartenance à un cycle, la cohésion sémantique, la granularité des composants, etc. Ce sont des critères appliqués dans l'évaluation des packages mais qui s'appliquent aussi parfaitement au cas des composants.

Néanmoins, l'introduction d'une multitude de nouveaux critères pose de nouveaux problèmes. En effet, certains de ces critères sont en contradiction entre eux. Par exemple, un système sans cycle entre composants implique un système avec peu de composants contenant beaucoup de classes et donc des composants peu cohésifs. Au contraire, une forte cohésion tend vers beaucoup de petits composants et donc la multiplication des cycles. Ce problème est illustré par la Figure 3.4 ou la cohésion d'un composant correspond au nombre d'appels entre les classes d'un composant pondéré par la taille du composant et les cycles d'un composant le nombre de cycles intercomposant dans le quels il est impliqué.

De plus, ces critères peuvent avoir des domaines de définition complètement différents. Par exemple, le couplage entre deux composants est souvent défini sur $\mathbb{N}^+$ alors que la cohésion sémantique entre deux composants peut l'être sur $[0, 1] \in \mathbb{R}$ [CKK08]. Ainsi, il faut trouver une fonction F qui à partir de n fonctions $f_1 \subset D_1, f_2 \subset D_2, ..., f_n \subset D_n$, définie sur n domaines $D_1, D_2, ..., D_n$ et soumise à n pondérations $\alpha_1, \alpha_2, .., \alpha_n$:

$$F(A) = F'(\alpha_1 * f_1(A), \alpha_2 * f_2(A), ..., \alpha_n * f_n(A))$$

Au vu de tous les problèmes soulevés, trouver une telle fonction semble particulièrement mal aisé. De plus, la pondération des critères peut varier d'une application à une autre [vDB11], ce qui ajoute encore un niveau de difficulté.

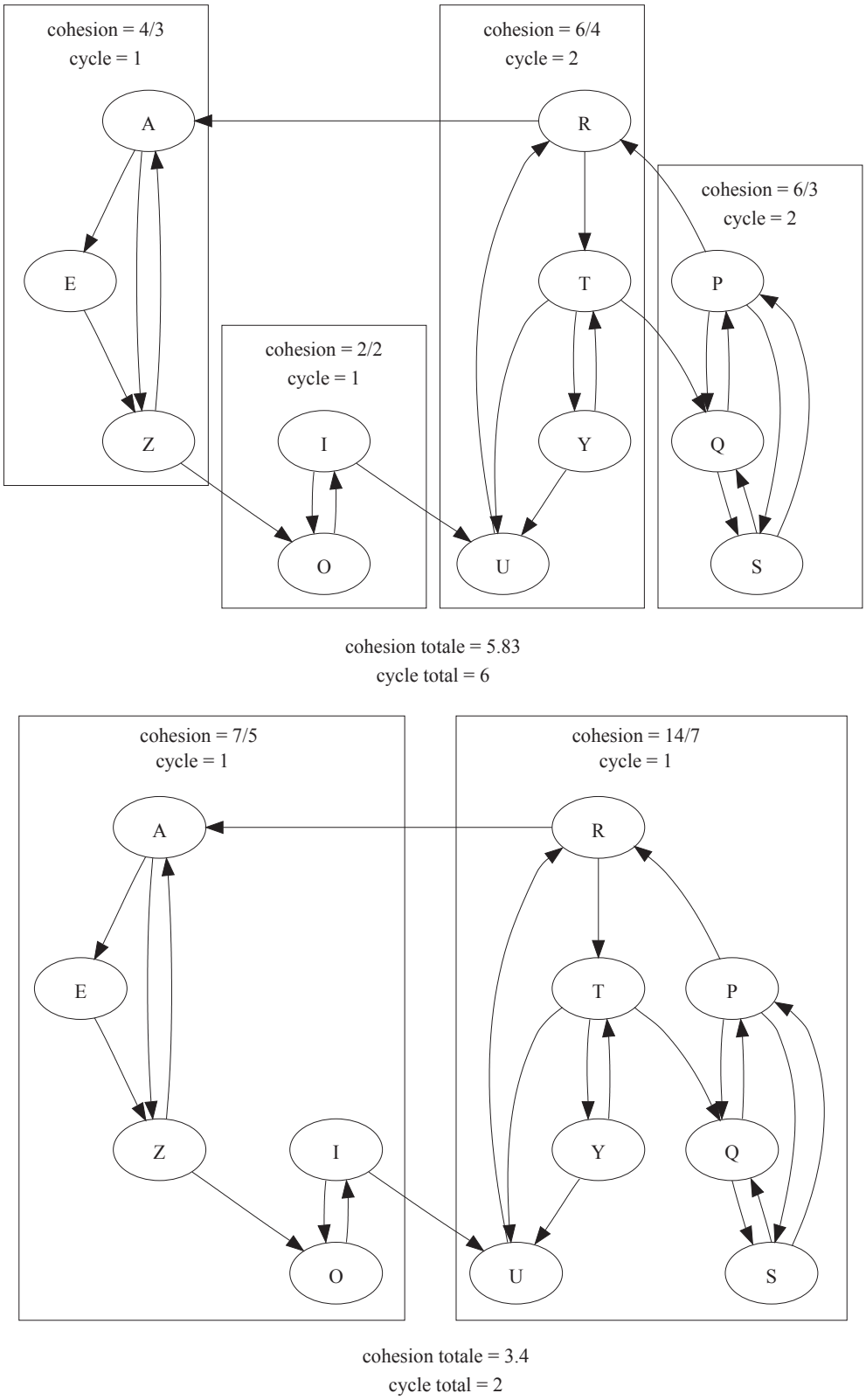


Figure 3.4 – Exemple de deux décompositions

3.7.4 Projection vers un modèle de composants concret

Après avoir identifié les composants d'un système, il peut être intéressant de les projeter vers un modèle concret pour effectuer une réingénierie de l'application OO sous la forme d'une application à base de composants. A ma connaissance, aucune approche ne le propose. Seuls Washizaki et al. [WF05] permettent d'extraire certains composants de l'application pour les réutiliser dans de nouvelles applications. Mais cette approche ne permet pas d'effectuer une réingénierie de l'application OO cible en application à base de composants.

D'abord, il faut identifier toutes les dépendances du composant. Sil est facile d'identifier toutes les dépendances liées aux liens d'héritage, cela se complique quand il s'agit des appels de méthode vers des classes extérieures au composant. Or l'oubli d'une seule dépendance peut compromettre l'intégrité de la nouvelle application.

Puis il faut définir les interfaces requises et fournies et fournir un mécanisme de gestion de ces dépendances/interface. Cela implique sûrement de restructurer une partie du code (par exemple, les interfaces requises) ou la mise en place d'un mécanisme complexe pour gérer les interfaces des composants. Ainsi, un composant projeté dans un modèle concret ne se résume pas simplement à un ensemble de classes identifiés.

Enfin, après avoir projeté les composants vers un modèle concret, il est normal d'espérer que ces composants deviennent des entités réutilisables. Or le contexte de leur identification peut être un obstacle à leur réutilisation. Prenons le cas des dépendances construites à partir des cas d'utilisation. Les cas d'utilisation sont définis dans le contexte précis de l'application. Ainsi, les composants qui en résultent sont liés à ce contexte. De ce fait, ces composants ne pourront pas être forcement réutilisé tel quel dans un autre contexte (application).

3.7.4.1 Résume des différents problèmes

Finalement, les différents problèmes relevés et qui peuvent biaiser le processus d'identification, peuvent être résumés de la façon suivante :

- **Fiabilité de la documentation** : généralement elle n'est pas complète ou/et mise à jour.
- **Fiabilité des graphes d'appel statique** : à cause des mécanismes objet, les graphes d'appel statique peuvent être incomplets ou/et incorrects.
- **Fiabilité des traces d'exécution** : leur nature dynamique fait que l'on ne peut garantir la couverture totale de l'application cible par les traces d'exécution.
- **Critère d'évaluation d'une ABC** : ils sont multiples et non limités au couplage et à la cohésion.
- **Evaluation d'une solution** : dans le cas où l'on utilise de multiple critères, il est très difficile de les composer dans une fonction unique d'évaluation de la qualité d'une ABC.
- **Identification de l'architecture à base de composants** : cette étape, très peu traitée dans la littérature, reste non triviale.
- **Nature OO du système cible** : certains "bons" patrons de conception dans une application OO de qualité peuvent gêner l'identification d'une ABC. De même, les *bad smell* de l'application peuvent bruiter le processus d'identification d'une ABC.

L'objectif de ce travail de thèse est de répondre à ces problèmes. Ainsi, dans la partie suivante, je présente un processus d'identification d'ABC qui essaye de répondre au maximum à ces problèmes.

Troisième partie

Contribution

Cette partie présente en détail la contribution de ma thèse : un processus semi-automatique d'identification d'une architecture à base de composant dans une application orientée objet. Dans le chapitre 4 une présentation générale de ce processus est donnée. Le chapitre 5 présente en détail les méthodes et algorithmes utilisés pour obtenir les données qui guident ce processus. Le chapitre 6 présente successivement et en détail les trois méthodes d'identification des composants développés, l'étape de raffinement manuelle des composants et enfin l'étape d'identification des interfaces des composants.

CHAPITRE 4

PRÉSENTATION GÉNÉRALE DE L'APPROCHE

Dans ce chapitre, je donne une présentation générale du processus semi-automatique d'identification d'une architecture à base de composants dans une application orientée objets, développé dans le cadre de cette thèse. Ce processus a été publiés dans [ASSF11]. Cette présentation est faite suivant les cinq critères présentés dans la section 3.1.

4.1 Approche

La figure 4.1 donne un aperçu du processus d'identification d'une architecture à base de composants (ABC) développé dans le cadre de cette thèse. Ce processus prend en compte les problématiques soulevées dans la section 3.7 et peut être décrit en fonction des critères présentés dans la section 3.1 :

- Ce processus, défini comme un problème de partitionnement, est global : chacune des classes d'application du système cible est assignée à un composant.
- L'identification d'une ABC est guidée par des données dynamiques (trace d'exécution). Quand cela est nécessaire, des données statiques (graphe d'appel statique) sont utilisées afin de compléter les résultats obtenus avec les données dynamiques.
- Trois méthodes d'identification de composants sont explorées.
- Ce processus est semi-automatique : les traces sont obtenues en exécutant manuellement les cas d'utilisation. De plus, une phase (facultative) de raffinement collaboratif est proposée au concepteur.
- Le modèle de composant utilisé est abstrait et couvre toutes les notions importantes du paradigme composant. Les ensembles de classes identifiés sont projetés dans ce modèle afin d'obtenir l'ABC de l'application cible.

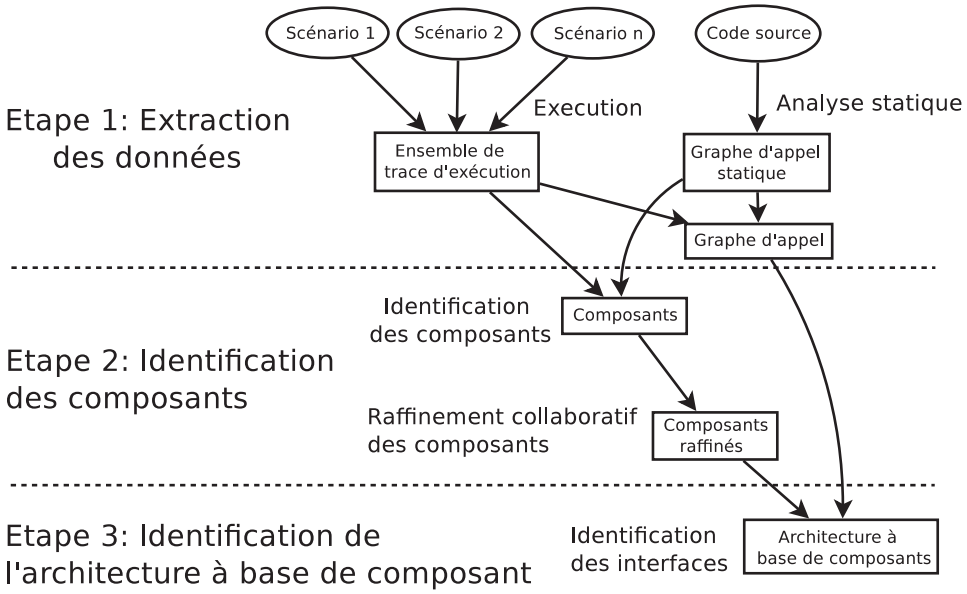


Figure 4.1 – Processus d'identification d'une architecture à base de composants

4.2 Processus global

Dans la section 3.7, j'ai présenté l'identification des composants comme un problème de partitionnement de l'ensemble des classes d'application du système cible. Formellement, identifier l'ensemble des composants d'une application consiste à trouver une partition $P(Cl) = \{C_1, C_2, .., C_n\}$ de taille n où Cl est l'ensemble des classes d'application du système cible et C_i un composant. La partition $P(Cl)$ doit respecter les propriétés de cohérence et de complétude :

- $\bigcup_{0<i\leq n} C_i = Cl$ (complétude)
- $\forall i, j | i \neq j, C_i \cap C_j = \emptyset$ (cohérence)

La propriété de complétude garantie que toutes les classes de Cl font partie d'un composant, la propriété de cohérence assure qu'aucune classe n'appartient à deux composants.

Ainsi, le processus d'identification d'une ABC que je propose considère toutes les classes du système cible : Il est donc global.

Il faut aussi noter que ces deux propriétés sont obligatoires si l'on veut projeter l'ABC obtenue vers un modèle concret. En effet, si une classe est manquante dans l'application réingéniéré, celle-ci ne fonctionnera pas ou, du moins, ne pourra pas fournir toutes les fonctionnalités. De plus, partager une classe entre plusieurs composants rend la réingénierie de l'application beaucoup plus complexe et augmente ainsi la possibilité d'introduire des erreurs ou encore de dégrader la qualité de l'application. En effet, les classes appartiennent à la même application. De ce fait, toutes les instances d'une même classe partagent la même définition de la classe : si je permets l'appartenance d'une même classe à plusieurs composants, cela créerait un conflit entre les deux espaces de définition de la classe. Or les deux espaces de définition doivent être distincts pour des raisons d'indépendance entre composants.

4.3 Données utilisées

Dans la première étape de mon processus (voir Figure 4.1), les données obtenues à partir de l'application cible pour l'identification d'une ABC sont de deux sortes : dynamique et statique. L'acquisition des deux types de données est décrite dans le chapitre 5.

Données dynamiques : comme il a été introduit dans la section 3.1, les dépendances extraites par une analyse statique du code source sont souvent utilisées dans les travaux existants. Or ces dépendances conduisent à une solution conforme à la conception orienté objets du système. Pour éviter ce problème, je propose de revenir à un niveau où les concepts OO n'influencent pas la conception architectural et ainsi la future ABC. Ainsi, la composition est guidée par les exigences fonctionnelles. Par conséquent, le processus d'identification des composants est guidé par les traces d'exécution obtenues à partir des cas d'utilisation de l'application cible. Les traces d'exécution capturent les dépendances entre les classes, lorsque celles-ci collaborent pour fournir une fonctionnalité particulière du système, fonctionnalité décrite par un cas d'utilisation. Ainsi, l'identification des composants est vue comme le regroupement des classes présentées dans les traces d'exécution conformément à leur interaction dans les cas d'utilisation.

De plus, utiliser des traces d'exécution permet d'éviter les problèmes liés à d'autres sources de données tels que la non mise à jour de la documentation (voir Section 3.7).

Données statiques : hélas, comme nous l'avons vu dans la section 3.7, il est difficile de garantir la couverture de toutes les classes de l'application cible avec les traces d'exécution si celles-ci ne sont pas complètes. Seules les classes présentes dans les traces interviennent dans le partitionnement. Or ce processus d'identification est global. Dans ce cas, les composants précédemment identifiés avec ces traces d'exécutions sont complétés à l'aide de données statiques : un graphe d'appel statique.

4.4 Identification des composants

Pour l'identification des composants (étape 2 de la figure 4.1), je propose trois méthodes différente développées successivement. Chaque nouvelle méthode vient résoudre des problèmes identifiés durant la validation de la méthode précédente.

Ces trois méthodes ont toutes en commun l'utilisation de données dynamiques pour identifier les principales classes de chaque composant de l'application cible (appelées noyaux des composants), puis l'achèvement des composants identifiés en utilisant des données statiques. En réalité, l'étape d'achèvement des composants n'est nécessaire que si le jeu de cas d'utilisation, utilisé pour obtenir les traces d'exécution, n'est pas complet et ne couvre pas toutes les classes de l'application.

La première méthode d'identification des composants utilise un treillis de Galois et une heuristique spécialement développée afin de déterminer les composants de l'application cible. La seconde méthode utilise deux meta-heuristiques : un algorithme générique suivi de l'algorithme *recuit simulé*. Ces deux méthodes utilisent le rapport cohésion/couplage pour identifier les composants. A ce niveau, l'originalité de mon approche réside dans l'utilisation de données dynamiques. Enfin, la troisième méthode, la plus aboutie, s'appuie sur le principe de sélection multi-critères. En effet, avec cette méthode quatre critères sont utilisés pour évaluer une solution : couplage, cohésion, granularité des composants et nombre de cycles entre les composants. Contrairement aux méthodes

précédentes, elle fournit un ensemble de solutions et non une solution unique. Dans cet ensemble, le concepteur choisit la solution qui lui convient le mieux. La troisièmement méthode utilise un algorithme génétique multi-critère basé sur le principe d'optimun de Pareto. (NSGA-II)

4.5 Processus semi-automatique

Le processus proposé est semi-automatique et cela, pour deux raisons. Premièrement, la principale source de données utilisée, les traces d'exécution, est obtenue en exécutant manuellement les cas d'utilisation de l'application cible. Secondement, la solution proposée par l'étape 2 n'est pas forcement parfaite. En effet, comme nous l'avons vu dans la section 3.7, l'application cible peut contenir des *bad smell* qui gênent le processus d'identification. De plus, deux des méthodes développées pour identifier les composants utilisent des meta-heuristiques. Ces algorithmes ne trouvent pas forcément la meilleure solution, mais une solution proche. Ainsi, une étape optionnelle de raffinement est proposée à l'utilisateur de l'approche. A partir d'informations sur les composants identifiés tels que leur couplage ou leur cohésion, l'utilisateur de l'approche peut manuellement modifier les composants identifiés.

4.6 Modèle de composants utilisé

Avant de pouvoir projeter les composants génériques (ensembles de classes) identifiés vers le modèle de composant, il faut définir leurs interfaces puis les identifier. Ainsi, les interfaces fournies (respectivement requises) par un composant sont les méthodes appelées à partir (respectivement vers) des classes appartenant à d'autres composants. Le modèle de la figure 4.2 formalise la projection des composants "objet" en proposant une projection du paradigme objet vers le modèle de composants utilisé dans le cadre de cette thèse.

La projection de ces ensembles de classe dans le modèle de composants est faite à l'étape 3 (voir Fig. 4.1) : les services requis et fournis de chaque composant sont iden-

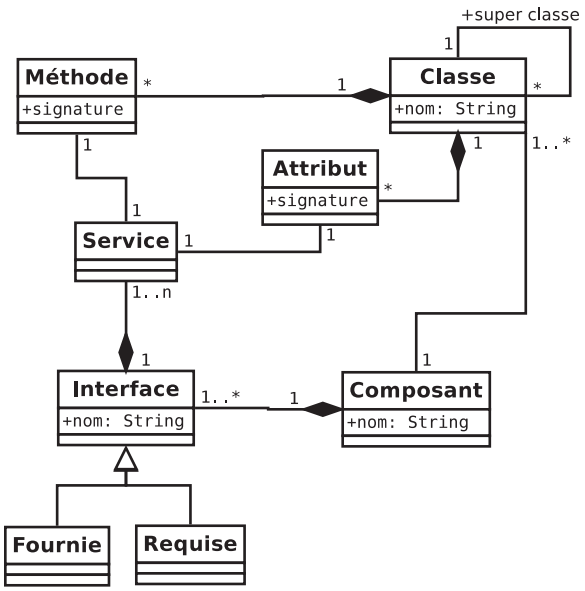


Figure 4.2 – Modèle de composants dans une application OO

tifiés puis regroupés dans une ou plusieurs interfaces. Ainsi, nous obtenons une projection de l'application cible OO dans le paradigme composant : une architecture à base de composant.

CHAPITRE 5

CAPTURE DES DÉPENDANCES ENTRE CLASSES

Mon processus d'identification se base sur les dépendances entre classes. Dans ce chapitre je présente plus en détail les deux types de données utilisées pour identifier un ABC (Section 5.1 et 5.2), ainsi que les différentes transformations appliquées sur ces données pour les utiliser (Section 5.3).

5.1 Capture dynamique des dépendances

L'analyse dynamique permet de capturer différents types d'informations pendant l'exécution d'un programme. Ces informations peuvent être l'utilisation des ressources (mémoire, processeur,...), les classes/méthodes utilisées par le programme ou encore la succession d'appels entre les méthodes.

Comme nous l'avons vu dans le chapitre 4, les traces d'exécutions sont la source principale de dépendance entre classes utilisées dans mon approche. Leur capture est présentée en détail ci-dessous.

Capture des dépendances

La capture des dépendances se fait en deux étapes : i) définition d'un ensemble de cas d'utilisation, i) capture des traces d'exécution associées à ces cas d'utilisation.

i) Définition d'un ensemble de cas d'utilisation : par définition, l'analyse dynamique ne peut pas garantir la couverture totale des fonctionnalités du système et donc des dépendances entre classes associées à ces fonctionnalités. En effet, l'analyse dynamique fournit uniquement les dépendances qui ont été mises en oeuvre lors des exécutions considérées. Ainsi, pour couvrir le maximum de dépendances, les traces sont obtenues en exécutant tous les cas d'utilisation du système cible. Ces cas d'utilisation peuvent être obtenus de la documentation du système et/ou proposés par un utilisateur de l'applica-

tion. Il est aussi possible de se baser sur les cas de test, ceux-ci sont sensés couvrir les différentes possibilités d'exécution.

ii) Capture des traces d'exécution : par défaut, on ne capture que les appels entre méthodes. Cependant, il est possible d'enrichir les traces avec d'autres types d'information dynamique tels que la création de nouveaux objets, la valeur des paramètres passés aux méthodes, la valeur des attributs d'un objet, etc. La seule limite à ces ajouts est leur coût (en temps et en espace). En effet, la capture des appels de méthodes se fait déjà à un coût non négligeable et considérer ces autres informations peut alourdir ce coût significativement. Ainsi, je ne capture que les appels entre méthodes et les accès en lecture/écriture sur les attributs.

Ainsi, les traces d'exécution sont obtenues en capturant les appels entre méthodes durant l'exécution d'un scénario spécifique à un cas d'utilisation. Chaque fil d'exécution (thread) créé durant l'exécution produit une trace. Les noeuds sont étiquetés avec la signature de la méthode appelée et le type dynamique de l'objet sur qui est invoquée la méthode.

Par exemple, l'exécution du code de la figure 5.1 donne la trace d'exécution de la figure 5.2d.

5.2 Capture statique des dépendances

Comme nous l'avons vu dans la section 3.1, l'analyse statique permet d'obtenir de nombreuses informations sur l'application. Dans la majorité des cas, l'information est complète, correcte et non bruitée.

Dans le cas des dépendances liées aux appels entre méthodes obtenues sous la forme de graphe d'appels, un certain nombre de problèmes se pose. En effet, pour construire ce graphe, il est nécessaire de connaître l'ensemble des méthodes qui peuvent être invoquées par chaque site d'appel de l'application cible (c'est à dire, les cibles d'invocation). Or, les mécanismes objet tels que l'invocation de méthode ou le chargement dynamique de classe ne sont pas faciles à simuler avec un algorithme et complexifie

ainsi le calcul du graphe d'appel de l'application [AVDS10].

Prenons l'exemple de la méthode la plus simple (*Declaring Target* (DT)) qui est la méthode la plus utilisée pour construire des graphes d'appels dans les outils de calcul de métriques. Pour chaque site d'appel de la méthode m d'une des classes que l'on considère, on crée un arc entre la méthode m et la cible déclarée du site d'appel. Or, comme on peut le voir dans la figure 5.2a correspondante au code de la figure 5.1, le graphe d'appels construit avec cette méthode est inexact car la méthode $useA$ n'appelle jamais la méthode $A.m$.

Pour éviter ce type de problème de construction du graphe, il faut utiliser des algorithmes de construction plus sophistiqués. La suite de cette section se focalise sur la construction du graphe d'appel avec de tels algorithmes.

5.2.1 Construction efficace du graphe d'appels

La construction du graphe d'appels a été largement étudiée dans la communauté d'analyse de programmes. Beaucoup de techniques ont été développées pour calculer des graphes d'appels statiquement à partir du code source.

Différents algorithmes construisant des graphes d'appels statiques ont été proposés. Ces algorithmes ont des précisions différentes. La construction d'un graphe d'appels précis exige des analyses sophistiquées qui sont coûteuses (en temps et en espace) tandis que les analyses les moins précises peuvent être facilement effectuées. L'algorithme de construction de graphe d'appels le plus basique est *Class Hierarchy Analysis* (CHA) [DGC95]. CHA considère seulement la hiérarchie des types pour calculer l'ensemble des cibles possibles associé à un site d'appel. En conséquence, il effectue une approximation grossière des cibles des sites d'appels, mais il a l'avantage d'être très peu coûteux. *Rapid Type Analysis* (RTA) [BS96] est presque identique à CHA, mais affine ses résultats en considérant l'ensemble des types d'objets qui peuvent être instanciés. L'observation clé faite par RTA est qu'un objet ne peut pas être un receveur pour un appel de méthode si aucun objet de son type n'a été instancié dans le programme. *Variable Type Analysis* (VTA) [SHR+00] est une simple analyse des flux de données qui suit chaque référence d'un objet (par exemple une variable) et maintient l'ensemble des types d'objets qu'il peut

s

```
static void main() {
  B b1 = new B();
  C c = new C();
  useA(b1);
  useB(c);
}

static void useA(A a) {
  a.m();
}

static void useB(B b2) {
  b2.m()
}
```

```
class A {
  void m() {...}
}

class B extends A {
  void m() {...}
}

class C extends B {
  void m() {...}
}

class D extends B {
  void m() {...}
}
```

Figure 5.1 – Exemple pour la construction des graphes d'appels

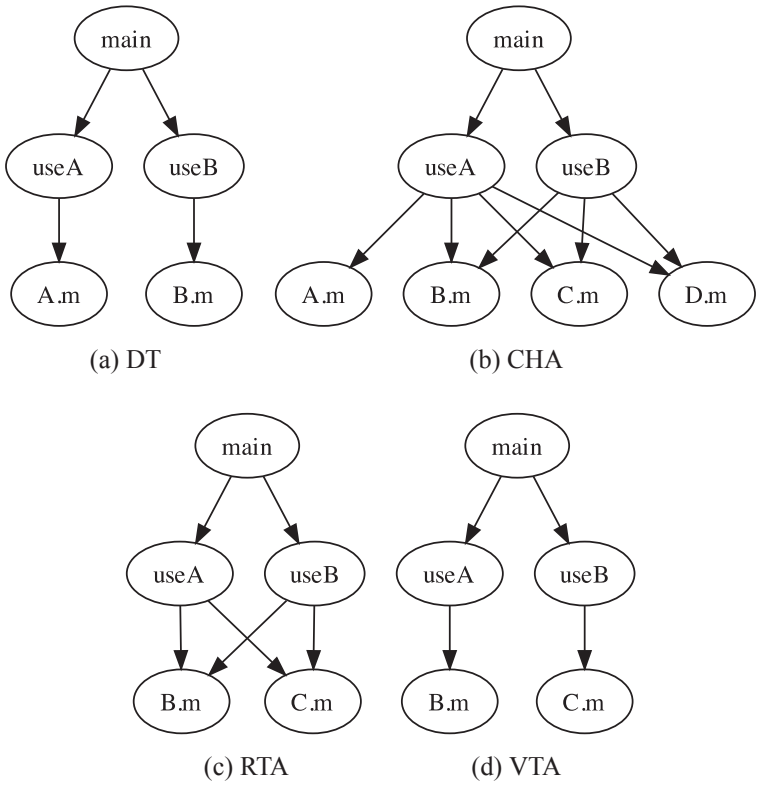


Figure 5.2 – Graphes d'Appels de l'exemple de la Figure 5.1

contenir. Ces informations sont utilisées pour réduire les cibles possibles pour chaque site d'appels. Il existe également d'autres techniques plus sophistiquées pour construire des graphes d'appels (par exemple. [GDDC97, TP00]), mais elles ne sont pas explorées ici faute d'intérêt relatif pour l'identification des composants. Je vais dans ce qui suit illustrer chacune des techniques avec l'exemple de la figure 5.1.

Exemples Dans l'exemple de la figure 5.1, La méthode `main` crée deux objets, un du type `B` et l'autre du type `C`, avant d'appeler les méthodes `useA` et `useB` avec ces objets en paramètre. L'inspection manuelle révèle facilement que la méthode `useA` appelle toujours la méthode `m` définie dans la classe `B`. En revanche, la méthode `useB` appelle toujours la méthode `m` de `C`. En utilisant l'algorithme DT, on obtient le graphe d'appels de la Figure 5.2a. Ce algorithme n'est pas conservateur. En effet, le graphe d'appels qu'il construit n'inclut pas le comportement réel du programme. Le graphe d'appels construit avec CHA est donné dans la Figure 5.2b. Ce graphe d'appels est conservateur, mais imprécis. En effet, l'algorithme considère que le site d'appel `a.m()` dans la méthode `useA` peut être lié à toutes les implémentations de `m`. De même, l'algorithme considère que l'appel `b2.m()` peut potentiellement invoquer l'implémentation de `m` dans la classe `B`. RTA peut améliorer la précision des graphes d'appels en observant qu'il n'y a pas d'objet de type `A` ou `D` qui est créé. Par conséquence, RTA considère seulement les implémentations de `m` dans les classes `B` et `C`. Néanmoins, la Figure 5.2c met en évidence que RTA n'est pas capable d'identifier le fait que chaque site d'appels à une cible unique. Finalement, VTA peut suivre les flux de types à travers les variables et attributs. Ainsi, le graphe d'appels construit est correct (Figure 5.2d).

5.2.2 Chargement dynamique de classes

Le chargement dynamique de classes peut aussi affecter la construction du graphe d'appels. En effet, certaines parties du code, chargées dynamiquement durant l'exécution du programme, ne peuvent être déterminées par les analyses statiques. Ainsi, les analyses statiques sont forcées de faire des suppositions conservatrices quant aux classes qui peuvent être chargées dynamiquement. Je considère que n'importe quelle classe d'ap-

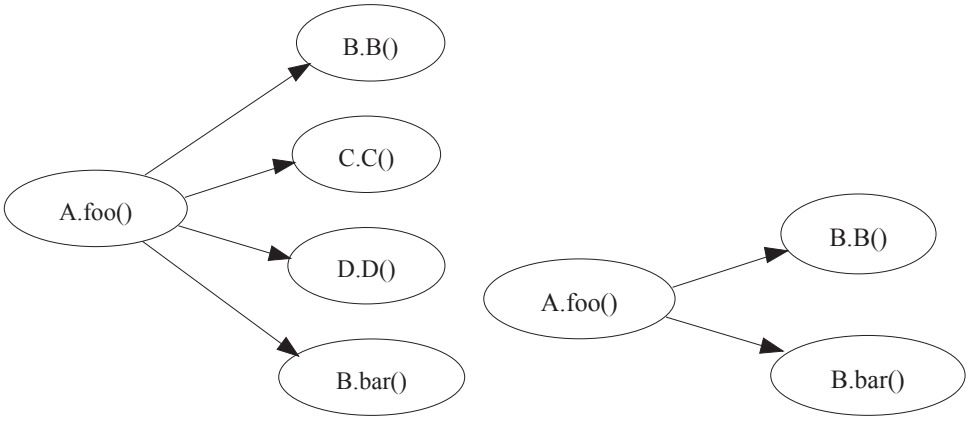
plication (classe qui n'appartient pas à une librairie) peut être potentiellement chargée durant l'exécution. Cela est presque toujours suffisant, même si, en pratique, cela ne garantit pas la supposition conservatrice : certains appels de méthodes "réel" peuvent toujours être ignoré.

```
class A {
  void foo() {
      Class c = Class.forName("B");
      MyClass obj = (B)c.newInstance();
    obj.bar(); // Use the object
      ...
  }
}
```

Figure 5.3 – Usage typique du chargement dynamique de classe en Java

Traiter le chargement dynamique de classes permet une meilleure capture du comportement du programme, mais ajoute une source d'imprécision en raison de la façon dont le chargement dynamique de classes est typiquement utilisé en Java. La Figure 5.3 illustre un scénario typique d'utilisation. Premièrement, une référence vers un objet `java.lang.Class` est obtenue durant d'exécution en utilisant le nom de la classe que l'on veut charger (ex. `String`). Deuxièmement, une instance de la classe nouvellement chargée est créée en utilisant la méthode `Class.newInstance`. Enfin, l'objet est "casté" avec son type réel pour être utilisé. A cause de la nature conservatrice des algorithmes de construction de graphe d'appels, l'ensemble des constructeurs sans arguments peut être potentiellement invoqué par l'appel de `newInstance`. Cela aura pour effet de créer des appels entre la méthode `A.foo()` et l'ensemble des constructeurs sans arguments. Pour résoudre ce problème, j'élague le graphe d'appels après sa construction. Tous les arcs provenant des sites d'appels qui correspondent aux méthodes `Class.forName` et `Class.newInstance` sont supprimés.

Par exemple, si l'on considère que l'application formée des classes d'application `A,B,C` et `D`, le code de la figure 5.3 donne, avec l'algorithme VTA et la gestion du chargement dynamique, le graphe d'appels de la figure 5.4a. Ce graphe d'appels contient



(a) Graphe d'appels avec gestion du chargement dynamique de classe

(b) Graphe d'appels élagué

Figure 5.4 – Graphes d'Appels de l'exemple de la Figure 5.3

2 appels qui n'ont jamais lieux : `C.C()` et `D.D()`. Au contraire, si l'on élague le graphe d'appels, on obtient le graphe d'appels de la figure 5.4b qui est correct.

Une étude de cas est présentée dans le chapitre 7 afin de déterminer l'influence de ces différents algorithmes sur le calcul des métriques de couplage.

Dans la suite de ce travail, les graphes d'appels statiques utilisés sont calculés avec l'algorithme VTA puis élagués.

5.3 Transformation

Les traces d'exécution capturées contiennent beaucoup d'informations et sont donc extrêmement volumineuses. Ainsi, selon le niveau de détail nécessaire, une transformation peut être appliquée aux traces d'exécution pour réduire leur taille. Sur les trois méthodes d'identification d'une ABC que je propose, seule la seconde méthode (voir Section 6.2) utilise les traces d'exécution telle quelles. Les autres méthodes utilisent des graphes d'appel entre classes construits à partir de ces traces. Un graphe d'appels entre classes est un graphe d'appels où les noeuds représentent des classes et non des méthodes et les arcs les appels de méthodes inter-classes.

La transformation d'une trace d'exécution en graphe d'appel entre classes se fait en deux étapes : i) transformation d'une trace en graphe d'appels entre méthodes, ii) transformation graphe d'appels entre méthodes en graphe d'appels entre classes.

De plus, pour chaque trace d'exécution transformée, un graphe d'appels entre classes différentes est obtenue. Or deux des méthodes d'identification utilisent un graphe unique obtenu dynamiquement et qui intègre les appels provenant de plusieurs exécutions. De même, la 3eme étape du processus d'identification d'une ABC utilise un graphe d'appels construit à partir de données statistiques et dynamiques. Ainsi, il faut pouvoir fusionner n graphes en un graphe unique.

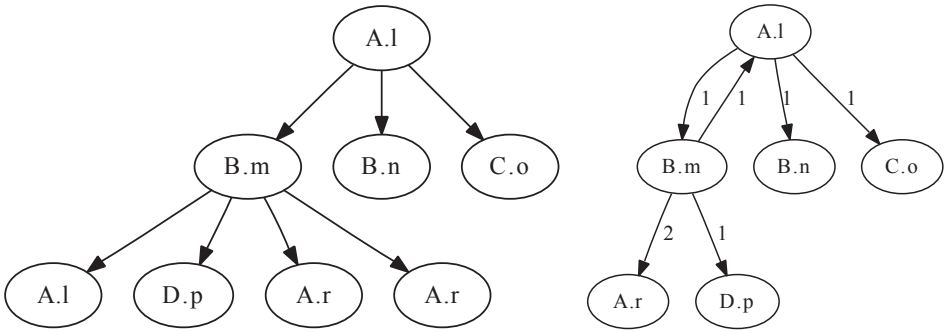
5.3.1 Traces d'exécution vers graphe d'appels

Une trace d'exécution est transformée en graphe d'appel de la façon suivante. Dans la trace, tous les noeuds étiquetés avec le même couple (classe,méthode) sont fusionnés. De plus, l'arc entre deux noeuds (l_1 et l_2) du graphe est étiqueté par le nombre d'arcs entre l_1 et l_2 dans la trace d'exécution. Cette information sur le nombre d'appels entre méthodes est un avantage de l'analyse dynamique, puisqu'il est impossible de l'obtenir statiquement. Cette information est utilisée dans la première méthode d'identification des composants.

Par exemple, dans la figure 5.5, on peut voir la transformation de la trace d'exécution (a) en graphe d'appels (b). Les deux instances de $A.l$ dans la trace fusionnent en un noeud unique dans le graphe. De plus, dans le graphe d'appels, l'arc $< B.m, A.r >$ est étiqueté par le nombre appel entre les méthodes $B.m$ et $A.r$ dans le trace, c.a.d 2.

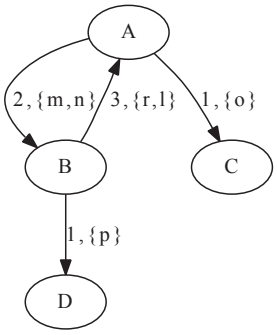
5.3.2 Graphe d'appels entre méthodes vers graphe d'appels entre classes

Le graphe d'appels entre méthodes est transformé en graphe d'appels entre classes de la façon suivante. L'ensemble des noeuds étiquetés avec la même classe est fusionné. Les arcs entre deux classes sont étiquetés par le nombre d'appels ainsi que l'ensemble des méthodes utilisées entre ces classes dans le graphe d'appels entre méthodes. Cette



(a) Trace exécution

(b) graphe d'appels



(c) graphe d'appels de classes

Figure 5.5 – Différents types de traces d'exécution

transformation peut être appliquée à un graphe d'appels obtenu statiquement ou dynamiquement.

Par exemple, dans la figure 5.5, on peut voir la transformation d'un graphe d'appels entre méthodes (b) en graphe d'appels entre classes (c). Les noeuds $B.n$ et $B.m$ fusionnent. De même, les arcs $< B.m, A.l >$ et $< B.m, A.r >$ fusionnent pour former l'arc entre B et A avec l'étiquette $(3, \{r, l\})$.

Au final, j'obtiens un graphe $G(V, E)$ où V est un ensemble de noeuds qui représente des classes et $E \in V \times V$ un ensemble d'arc qui représente des appels entre classe. Les arcs sont étiquetés par la paire $(nbFctCall, nbCall)$:

- $nbCall : E \longrightarrow \mathbb{Z}$ le nombre d'appel entre deux classe.
- $fctCall : E \longrightarrow P(M)$ ensemble des éléments appelés dans la classe cible de l'arc.

Avec M l'ensemble méthode/attribut/constructeur du système cible.

5.3.3 Fusion de graphes

Comme je désire produire un graphe d'appel pour un ensemble d'exécutions, je propose une méthode permettant de produire un tel graphe par fusion des graphes résultant de l'étape précédente.

La fusion de deux graphes d'appels entre classes se fait de la façon suivante. Soient $G_1(V_1, E_1)$ et $G_2(V_2, E_2)$, $G(V, E) = G_1 + G_2$, est obtenu suivant les règles :

1. $V = V_1 \cup V_2$
2. $E = E_1 \cup E_2$
3. $si\ e_1 =< x, y > \in E_1$, $e_2 =< x', y' > \in E_2$, $x = x'$ et $y = y'$ alors
   $e =< x, y > \in E, nbCall_e = nbCall_{e_1} + nbCall_{e_2}$ et
   $fctCall_e = fctCall_{e_1} \cup fctCall_{e_2}$

La troisième règle permet de fusionner deux arcs ayant les mêmes noeuds source et cible dans leurs deux graphes respectifs.

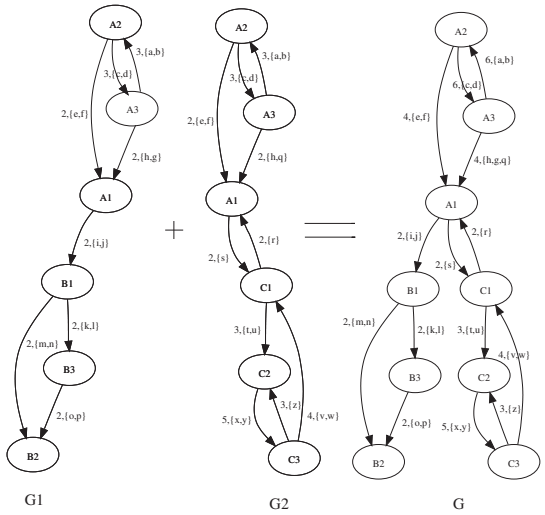


Figure 5.6 – Exemple de fusion de graphes

Par exemple, dans la Figure 5.6, les arcs $< A_3, A_1 >$ respectivement de G_1 et G_2 sont fusionnés dans l'arc $< A_3, A_1 >$ de G avec les valeurs suivantes :

$nbCall_{<A3,A1>_G} = nbCall_{<A3,A1>_{G_1}} + nbCall_{<A3,A1>_{G_2}} = 4$

$fctCall_{<A3,A1>_G} = fctCall_{<A3,A1>_{G_1}} \cup fctCall_{<A3,A1>_{G_2}}) = \{h, g, q\}$

De plus, il est facile de voir que les trois règles de l'opérateur de fusion (+) sont commutatives et associatives. Ainsi, la fusion des n graphes peut être faite en appliquant l'opérateur binaire de fusion dans n'importe quel ordre, sans que cela n'influence le résultat final.

CHAPITRE 6

IDENTIFICATION D'UNE ARCHITECTURE À BASE DE COMPOSANTS

Dans ce chapitre, je commence par présenter en détail trois approches d'identification des composants développées dans le cadre de ma thèse. Je présente ces approches en respectant l'ordre chronologique de leur mise au point. La troisième est la plus aboutie car elle tire les leçons des deux précédentes. Ensuite, les étapes de raffinement manuelle des composants et d'identification des interfaces des composants sont présentées.

6.1 Identification de composants avec un treillis de Galois

La première approche que j'ai étudié utilise un treillis de Galois pour identifier les composants de l'application cible. Elle utilise un graphe d'appels dynamiques de classes et un graphe d'appel statique de classes. Cette approche comprend 4 étapes (figure 6.1) :

1. Génération d'un ensemble de groupes candidats (avec un treillis de Galois). Chaque groupe candidat peut être un composant.

2. Dans l'ensemble des groupes candidats, sélection des groupes qui forment les noyaux des composants de l'application cible.

3. Raffinement des noyaux des composants sélectionnés.

4. Ajout des classes manquantes aux noyaux des composants raffinés.

Cette approche utilise deux critères pour évaluer la qualité d'un composant : le couplage avec les autres composants, qui doit être bas et la cohésion interne, qui doit être élevée. Les étapes de sélection et raffinement respectent la propriété de cohérence. La quatrième étape permet à cette approche de répondre à la propriété de complétude.

Cette approche a été publiée dans [ASS09].

Les sous-sections qui suivent détaillent ces 4 étapes.

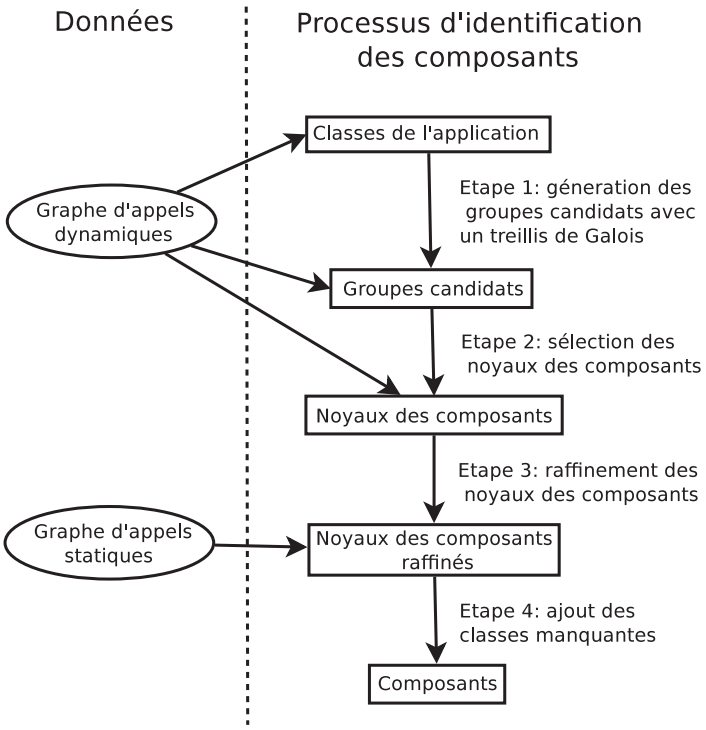


Figure 6.1 – Processus d'identification de composants

6.1.1 Génération des groupes candidats

Comme nous l'avons vu, l'identification des composants peut être vue comme un problème de partitionnement. Pour un ensemble de classe Cl, avec $|Cl| = n$ classes, il y a 2^n sous-ensemble possible. Certains des sous-ensembles sont significatifs : les classes formant ce groupe sont effectivement couplées dans le graphe d'appels. Mais la majorité n'est pas pertinente (les classes n'ont aucun lien entre elles). Pour trouver les ensembles significatifs, j'utilise l'analyse formelle de concepts ou treillis de Galois [GMM+95].

Définitions : Le treillis de Galois [GMM+95] permet de trouver les ensembles d'objets partageant les mêmes propriétés.

Soit deux ensemble finis S et S', une relation binaire $R \subseteq S \times S'$ et $P(S)$, $P(S')$ les ensembles de parties de S, respectivement S'. Chaque élément du treillis est une paire, noté (In, Ex), composée d'un ensemble $In \in P(S)$ et d'un ensemble $Ex \in P(S')$ vérifiant les deux propriétés suivantes :

- $Ex = f(In)$ ou $f(In) = \{x' \in S' \mid \forall x \in In, xRx'\}$ est l'ensemble de toutes les images des éléments de In par la relation R.
- $In = f'(Ex)$ ou $f'(Ex) = \{x \in S \mid \forall x' \in Ex, xRx'\}$ est l'ensemble de tous les antécédents des éléments de Ix par la relation R.

Le treillis de Galois G correspond à l'ensemble des paires (In, Ex).

Dans cette méthode, un treillis est calculé à partir du graphe d'appels dynamiques $G(V, E)$. Les ensembles S et S' sont égaux à V, l'ensemble des classes qui apparaissent dans les traces d'exécution (V est inclue dans Cl, l'ensemble des classes de l'application cible). La relation binaire R correspond à la relation $E : xRx \Leftrightarrow (x, y) \in E$. Deux classes de V sont en relation si il existe un arc entre elles dans G.

Le treillis de Galois donne les ensembles de classes qui sont en relation directe. En effet, dans la paire (In_i, Ex_i), Ex_i est l'ensemble des classes qui appellent toutes les classes de In_i. Le composant candidat correspond ici à l'ensemble $In_i \cup Ex_i$. Par exemple, le treillis de la figure 6.2, construit à partir du graphe d'appels de la Figure 5.6, donne les composants candidats suivants : $\{A1, A2, A3\}$, $\{C1, C2, C3\}$, $\{C2, C3\}$, $\{A1, A2, A3, C1$ $\{A1, C1, C3\}$, $\{A1, C1, C2\}$, $\{B1, A1, C1\}$, et $\{B1, B2, B3\}$.

Complexité : Soit n la taille de l'ensemble S et k la borne supérieur de $f(\{s\})$ tel que $\forall s \in S \; |f(\{s\})| \leq k$. Dans [GMM+95], Godin montre que la taille du treillis (le

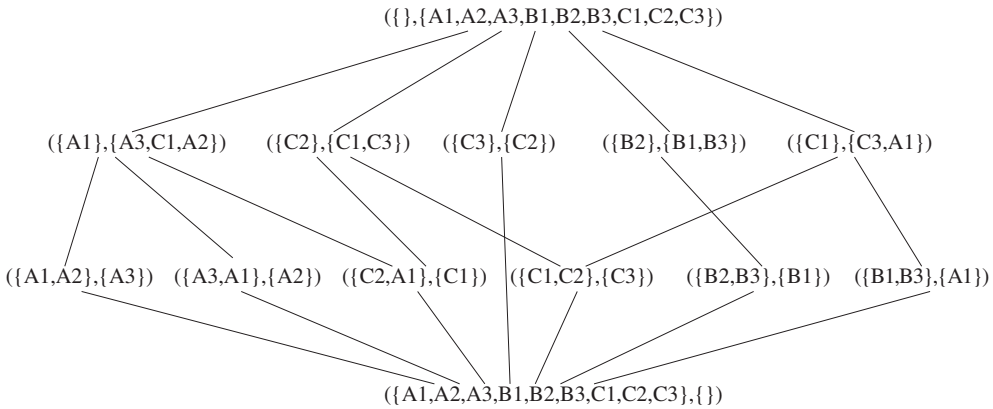


Figure 6.2 – Treillis de Galois de la Figure 5.6

nombre maximal de noeuds, noté nl) est bornée par $nl \leq 2^k n$.

Ici, n correspond au nombre de classes et k au nombre maximal de classes appelées par une classe. Par conséquence, la taille du treillis augmente linéairement avec la taille du programme en termes de nombre de classes. En effet, je conjecture que le nombre maximal de classes appelées par une classe dépend faiblement de la taille de programme.

6.1.2 Sélection des composants

Comme décrit ci-dessus, le treillis de Galois contient l'ensemble des groupes significatifs. Le but de cette étape est de sélectionner parmi les groupes candidats, le sous-ensemble qui maximise la fonction de qualité $evalComp(C_i)$ tout en satisfaisant la propriété de cohérence. Cette sélection est faite avec l'algorithme *Selection* (Algorithme 1).

La fonction d'évaluation $evalComp(C_i)$ d'un composant C_i est calculée en utilisant un graphe d'appel entre classes ainsi que les étiquettes $nbCall$ et $fctCall$ définies sur les arc de ce graphe (voir chapitre 5). $evalComp$ est normalisée entre 0 et 1, si $evalComp(C_i) = 0$ alors il n'y a aucune relation entre les classes de C_i. Au contraire, si $evalComp(C_i) = 1$, alors les classes de C_i sont toutes reliées entre elles. $evalComp$ est calculée comme la moyenne de la fonction $eval(c_j, C_i)$. $eval(c_j, C_i)$ évalue la contribution d'une classe c_j dans le composant C_i. Formellement :

$$evalComp(C_i) = \sum_{c_j \in C_i} \frac{eval(c_j, C_i)}{|C_i|}$$

avec $eval(c_j, C_i)$:

$$eval(c_j, C_i) = \frac{NC(c_j, C_i)}{NC(c_j, C_i) + \sum_{c_l \notin C_i} |fctCall_{<c_j, c_l>}|}$$

et $NC(c_j, C_i)$:

$$NC(c_j, C_i) = \sum_{c_k \in C_i} nbCall_{<c_j, c_k>}$$

Le raisonnement derrière la fonction $evalComp$ est le suivant. Les classes à l'intérieur d'un composant doivent être cohésives, c'est-à-dire, collaborer ensemble pour exécuter une fonction particulière. Pour exécuter cette fonctionnalité, les classes de ce composant ne doivent pas utiliser beaucoup de services extérieurs. Le degré de collaboration entre deux classes c_j et c_k dans C_i est capturé par le nombre d'appels entre elles ($nbCall_{<c_j,c_k>}$). Pour être en accord avec le principe de composition, le couplage entre une classe c_j appartenant à un composant C_i et une classe c_l n'appartenant pas à C_i, est capturé par le nombre d'éléments différents (les services) de c_l qui sont appelés par c_j ($fctCall_{<c_j,c_l>}$).

L'algorithme *Selection* (Algorithme 1) est un algorithme de type glouton. Il est basé sur $evalComp$ et utilise les variables et fonctions suivantes :

- GL : l'ensemble des groupes candidats du treillis.
- Cs : l'ensemble des composants sélectionnés.
- ts : un paramètre qui définit le seuil de qualité d'un composant candidat.
- $sort(GL)$: une fonction qui trie les composants candidats en fonction de leur qualité.
- $pop(GL)$: une fonction qui retourne le meilleur composant.
- $update(GL, c)$: une fonction qui supprime les classes du composant sélectionné dans tous les composants candidats restants.

```
1  Algorithme : selection(GL, ts)
2  Cs := ∅
3  sort(GL)
4  co := pop(GL)
5  while evalComp(co) > ts and |GL| > 0 do
6  |   Cs := Cs ∪ {co}
7  |   update(GL, co)
8  |   sort(GL)
9  |   co := pop(GL)
10 end
11 return Cs
```

Algorithme 1: Algorithme Selection

L'algorithme $selection(GL,\ ts)$ sélectionne les composants qui ont une bonne cohésion et un couplage bas. À chaque itération, les composants candidats sont triés selon $evalComp$ (ligne 3 et 8). Le candidat avec la valeur la plus haute est alors choisi et est ajouté à l'ensemble des composants sélectionnés (ligne 6). Pour satisfaire la propriété de cohérence, toutes les classes de ce composant sont ensuite enlevées des candidats restants (ligne 7). Ceux-ci sont à nouveau triés après l'opération de mise à jour et une nouvelle itération est exécutée (ligne 8). L'algorithme termine quand l'une des deux conditions suivantes est atteinte : toutes les classes du programme sont déjà dans les composants choisis, ou la qualité de tous les candidats restants est en dessous du seuil de qualité ts.

L'application de l'algorithme de sélection sur les candidats du treillis de la Figure 6.2 fournit trois composants (voir Figure 6.3). Les trois itérations de l'algorithme correspondant à la sélection des trois composants sont récapitulées dans la Figure 6.4.

6.1.3 Raffinement des Composants

L'algorithme de sélection précédent peut mener à un optimum local. En effet, comme les classes sont enlevées des composants sélectionnés aux composants candidats restants, cette sélection peut impacter la qualité des composants restants et donc la qualité globale

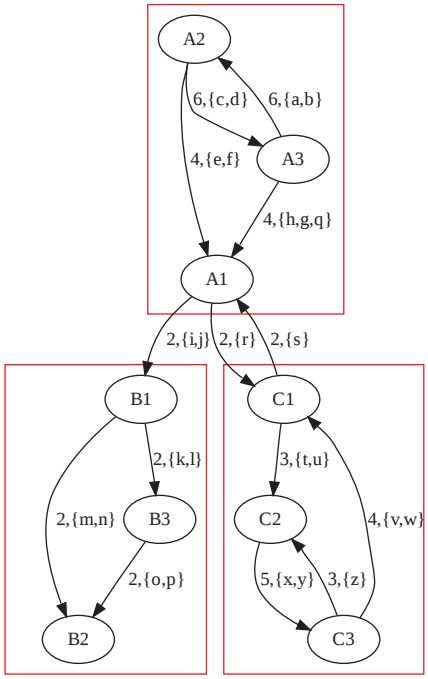


Figure 6.3 – Composants identifiés à partir du graphe d'appels dynamique

Itération 1	
composant	eval
$\{C1, C3, C2\}$:	0.94
$\{B2, B3, B1\}$:	0.9
$\{A2, C1, A1\}$:	0.83
$\{A2, C1, A1, A3\}$:	0.76
$\{C3, C2\}$:	0.61
$\{C1, C3, A1\}$:	0.45
$\{C1, A1, C2\}$:	0.42
$\{C1, A1, B1\}$:	0.32

Itération 2	
composant	eval
$\{B2, B3, B1\}$:	0.9
$\{A2, A1, A3\}$:	0.83
$\{A1, B1\}$:	0.21
$\{A1\}$:	0

Itération 3	
composant	eval
$\{A2, A1, A3\}$:	0.83
$\{A1\}$:	0

	Composants Sélectionnés	
$\{C1, C3, C2\}$	$\{C1, C3, C2\}$	$\{C1, C3, C2\}$
	$\{B2, B3, B1\}$	$\{B2, B3, B1\}$
		$\{A2, A1, A3\}$

Figure 6.4 – Exemple de sélection de composant

de la solution trouvée. Idéalement, mon algorithme devrait explorer toutes les possibilités et choisir la meilleure. Cependant, comme nous l'avons vu dans la section 6.1.2, le nombre des candidats peut être énorme (jusqu'à $2^k n$). Explorer toutes les possibilités revient à explorer toutes les permutations (jusqu'à $(2^k n)!$), ce qui est impossible en pratique.

Alternativement, j'ai décidé d'utiliser une étape de post-traitement, appelée *raffinement*. Cette étape modifie les composants choisis pour améliorer leur qualité. L'algorithme de raffinement (Algorithme 2) réutilise la fonction $eval$ pour évaluer la contribution d'une classe dans un composant particulier (voir la Section 6.1.2).

Voici les variables et fonctions utilisées par l'algorithme *Raffinement* :

- Cs : l'ensemble des composants sélectionnés.
- Cns : l'ensemble des classes qui ne sont pas incluses dans un composant après l'étape de sélection.
- $ts2$: un paramètre qui définit le seuil à considérer pour qu'une classe puisse être incluse dans un composant.
- $MaxIter$: le nombre maximum d'itération que peut faire l'algorithme.
- $nUClasses(Cs,\ ts2)$: une fonction qui retourne toutes les classes dont l'évaluation ($eval$ par rapport à leur composant) est inférieure au seuil $ts2$. Les classes retournées sont supprimées de leur composant.
- $addClass(Cs,\ c,\ ts2)$: Une fonction qui ajoute une classe c à un composant de l'ensemble Cs avec lequel il a le meilleur résultat avec $eval$. Si l'inclusion dégrade la qualité globale du composant, la classe n'est pas ajoutée. La fonction retourne un booléen pour notifier ou non le succès de l'inclusion.

```
1  Algorithme : raffinement(Cs, Cns, ts2, MaxIter)
2  iter := 0
3  Cns := nUClasses(Cs, ts2) ∪ Cns
4  while Cns ≠ ∅ and iter < MaxIter do
5  |   iter := iter + 1
6  |   for c ∈ Cns do
7  |   |   if addClass(Cs, c, ts2) then
8  |   |   |   Cns := Cns − {c}
9  |   |   end
10 |   end
11 end
12 return Cs
```

Algorithme 2: Algorithme Raffinement

Le principe de l'algorithme de raffinement est le suivant. D'abord, toutes les classes qui dégradent la qualité de leur composant respectif ($eval$ en dessous de $ts2$) sont enlevées de ce composant (ligne 3). Ensuite, pour chacune de ces classes, l'algorithme cherche un composant pour lequel elle serait le plus utile : on ajoute une classe à un composant si elle ne dégrade pas la qualité de ce composant selon $evalComp$ (ligne 7). Cette phase est réitérée jusqu'à ce que toutes les classes soient assignées à des composants ou que le nombre maximal d'itération soit atteint. Il est nécessaire de réitérer la phase d'ajout des classes pour la raison suivante : si initialement une classe c_j n'est pas utile ou dégrade la qualité d'un composant C_i, l'introduction d'une autre classe c_k dans C_i peut modifier positivement $eval(c_j,\ C_i)$ et $evalComp(C_i)$. Finalement, cet algorithme retourne l'ensemble des noyaux des composants de l'application. Cet ensemble respecte la propriété de cohérence.

6.1.4 Ajout des classes manquantes

Comme nous l'avons vu, les traces d'exécution ne peuvent garantir la couverture totale des classes du système cible. D'un point de vue fonctionnel, ces classes sont moins importantes, du moins pour les cas d'utilisation exécutés. Ainsi, elles ne devraient pas avoir une influence sur l'ABC final. J'ai donc décidé de ne les considérer que dans une seconde étape afin de respecter la propriété de complétude. Cette étape est guidée par

le graphe d'appels statique de l'application cible et l'algorithme de raffinement (Algorithme 2).

Cette étape se déroule de la façon suivante : chaque classe qui n'appartient pas un composant est ajoutée à l'ensemble Cns. Ensuite, l'algorithme de raffinement est relancé. L'évaluation de la qualité d'un composant C avec $evalComp(C)$ se fait sur le graphe d'appels statique. Ainsi, je suis sûr d'obtenir les dépendances entre les classes manquantes et les classes affectées à un composant dans les étapes précédentes. De plus, toutes les classes non affectées à un composant après l'exécution de l'algorithme sont des composants singleton.

Finalement, j'obtiens l'ensemble des composants de l'application cible. Cet ensemble de composants respecte les propriétés de cohérence et de complétude.

6.1.5 Etude de cas

Afin d'évaluer la pertinence de cette approche, deux études de cas simples ont été réalisées. La première porte sur un interprète du langage Logo, la seconde sur une application démonstration du framework JHotDraw.

Ces deux études de cas doivent permettre de répondre à la question suivante : **les composants identifiés sont-ils corrects ?** Je considère qu'un composant est correct s'il fournit une unique fonctionnalité et contient toutes les classes liées à cette fonctionnalité.

6.1.5.1 Interprète Logo

Logo est un langage de programmation créé pour l'apprentissage de la programmation et possède pour cela des instructions graphiques très faciles à utiliser. L'interprète a une interface graphique qui permet d'écrire le code et une fenêtre qui affiche le résultat des instructions graphiques. L'interprète Logo a été choisi pour deux raisons : i) sa taille (40 classes et 2 interfaces) me permet d'effectuer une analyse approfondie des résultats, ii) j'ai participé à son développement. Ainsi, il est facile d'évaluer la qualité de la solution proposée.

L'extraction des composants prend en entrée un graphe d'appels dynamique construit à partir de 26 traces d'exécution qui correspondent aux scénarios de 6 cas d'utilisation. Par exemple, les cas d'utilisation sont "création d'un fichier", "écriture de code dans l'éditeur", "interprétation du code", etc. Les traces d'exécution couvrent 39 classes du programme. Le graphe d'appels statique contient toutes les classes de l'application. Les paramètres ts et $ts2$ sont réglés à 0.7.

Résultat. Six composants ont été identifiés dans l'interpréteur Logo. Deux d'entre eux sont des singletons. Les quatre composants non singleton sont nommés avec la fonctionnalité qu'ils fournissent :

- **Parser** : ce composant contient toutes les classes du parseur du langage Logo. Deux des neuf classes affectées à ce composant ne participent pas à cette fonction : elles sont liées à l'interface graphique du composant.
- **Evaluator** : ce composant évalue le résultat du composant **Parser**. Il contient l'évaluateur du langage ainsi que les classes fournissant les fonctions non graphiques du langage.
- **Display** : ce composant fournit les fonctions graphiques de Logo. Il contient toutes les classes responsables de l'affichage graphique ainsi que la bibliothèque qui permet de les manipuler. Une classe a été attribuée à tort à ce composant et devrait être dans le composant **GUI**.
- **GUI** : ce composant fournit l'interface graphique du programme. Il n'est pas complet ; les deux singletons devraient se trouver dans ce composant, de même que les trois classes assignées aux composants **Display** et **Parser**.

A part le composant GUI, les composants identifiés sont complets et fournissent bien une fonctionnalité de haut niveau de l'application cible. Pour cette application, l'approche donne des résultats satisfaisants.

6.1.5.2 Application démo du framework JHotDraw

JHotDraw est un framework et non un programme. L'extraction des traces n'est pas faite à partir de JHotDraw mais à partir d'une application de démonstration qui utilise le framework. Le graphe d'appels dynamique est construit à partir de 26 traces d'exécution correspondant à 8 cas d'utilisation. Les traces d'exécution couvrent 204 classes du programme.

Résultat. Le processus d'identification génère 22 composants incluant au total 153 classes. 51 classes ne sont pas assignées à un composant. La majorité de ces 22 composants semble relier à une unique fonctionnalité. Par exemple, la création d'une figure, la sauvegarde d'une figure ou encore le chargement d'une figure. Cependant, dans de nombreux cas, les composants ont une granularité très fine. Ils pourraient être fusionnés entre eux pour former des composants de taille plus importante. Par exemple, les trois composant qui permettent de manipuler des figures pourraient être fusionnés dans un unique composant.

Cela peut être expliqué par le fait que le treillis de Galois ne peut que trouver des ensembles de classe en relation directe dans le graphe d'appels. Or pour fournir une fonctionnalité, les classes d'un composant ne sont pas toutes forcement en relation directe.

De plus, parmi les 51 classes non assignées à un composant beaucoup devraient fusionner avec un ou plusieurs composants.

Le résultat obtenu sur cette application n'est pas satisfaisant. En effet, la taille des composants est trop importante pour qu'ils puissent être identifiés par cette approche. De plus, beaucoup de classes ne sont pas affectées à un composant alors qu'elles le devraient.

6.1.6 Conclusion

Dans cette section, j'ai présenté la première méthode d'identifications de composant que j'ai développée. Celle-ci utilise des données dynamiques et statiques pour guider la

construction d'un treillis de Galois. Ce treillis permet d'obtenir les composants de l'application cible. Deux études de cas ont été réalisées. Elles montrent que cette méthode donne de bons résultats pour l'identification de composant contenant de peu de classes. Mais elle rencontre des problèmes quand la taille augmente. J'ai donc développé une autre approche qui résout ce problème. Je vais la présenter dans la section suivante

6.2 Identification de composants par conformité aux traces

Avec cette approche, le partitionnement de l'ensemble des classes du système cible est fait à l'aide de deux méta-heuristiques. Une méta-heuristique est définie par Osman et al. [OL96] comme : *"un processus itératif qui subordonne et qui guide une heuristique, en combinant intelligemment plusieurs concepts pour explorer et exploiter tout l'espace de recherche. Des stratégies d'apprentissage sont utilisées pour structurer l'information afin de trouver efficacement des solutions optimales, ou presque-optimales"*.

Les méta-heuristiques sont des stratégies qui permettent de guider la recherche d'une solution optimale, d'explorer l'**espace de recherche** efficacement afin de déterminer une solution (presque) optimale. Les algorithmes de type méta-heuristique vont de la simple procédure de recherche locale à des processus d'apprentissage complexes.

Dans ce cas, j'utilise deux algorithmes stochastiques itératifs qui progressent vers un optimum global par échantillonnage d'une **fonction objectif**. Ces deux algorithmes sont : un algorithme génétique [Hol75] (GA) qui est une recherche globale et l'algorithme *recuit simulé* [MRR+53] (SA), un algorithme de recherche locale.

La figure 6.5 décrit le processus d'identification des composants. La première étape identifie les noyaux des composants à partir des traces d'exécution. Cela est fait en utilisant une recherche hybride [KCLW08] qui combine les deux méta-heuristiques GA et SA. Cette étape est elle-même divisée en deux sous-étapes. L'étape 1a identifie les noyaux des composants avec l'algorithme GA. Ensuite ils sont raffinés par l'algorithme SA dans l'étape 1b. L'espace de recherche ainsi que la fonction objectif sont les mêmes pour ces deux algorithmes. La fonction objectif favorise les composants dont les classes

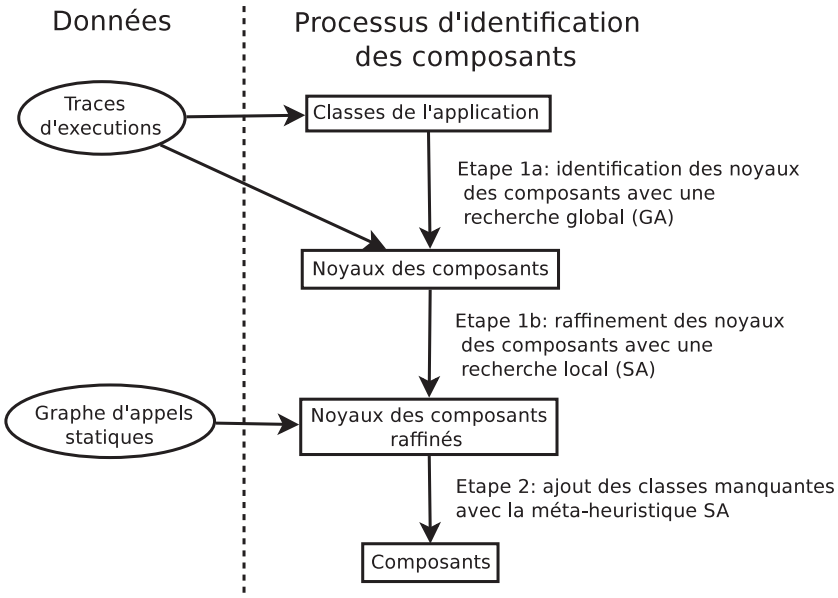


Figure 6.5 – Processus d'identification de composants

collaborent beaucoup entre elles dans les traces d'exécution tout en pénalisant les composants avec trop de couplage. La seconde étape ajoute les classes qui n'apparaissent pas dans les traces aux noyaux des composants. Cela est fait à partir d'un graphe d'appels statique et l'algorithme SA. L'espace de recherche ainsi que la fonction objectif sont très proches de ceux utilisés dans la première étape. Au final, nous obtenons l'ensemble des composants du système cible. Cet ensemble respecte les propriétés de cohérence et de complétude. Cette approche a été publiée dans [ASSV10]. Dans la suite, je vais présenter en détail le processus décrit dans la figure 6.5, étape par étape.

6.2.1 Identification des noyaux des composants

Cette sous-section présente l'étape d'identification des noyaux des composants. En premier, je vais présenter l'espace de recherche et la fonction objectif qui évalue les éléments de cet espace. Les deux algorithmes GA et SA seront ensuite présentés en détail.

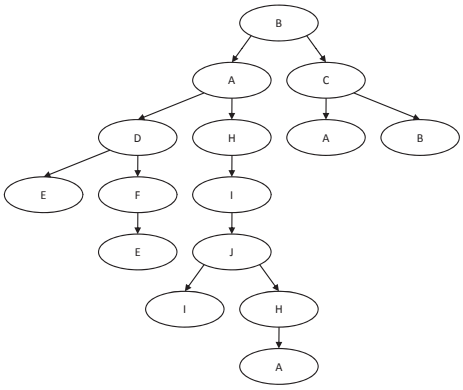


Figure 6.6 – Exemple de traces d'exécution

6.2.1.1 Encodage des solutions

Une solution pour les deux algorithmes peut être n'importe quelle partition des classes présente dans les traces d'exécution. Cette partition doit vérifier les propriétés de cohérence et de complétude. Chaque élément de cette partition représente un noyaux de composant de l'application cible. Dans l'exemple de la Figure 6.6, la trace d'exécution contient les classes suivantes : $A, B, C, D, E, F, H, I,\ J$. Une solution possible est : $\{\{A, C, D\}, \{E, J\}, \{H, I, B, F\}\}$.

Ainsi, l'espace de recherche des deux algorithmes correspond à l'ensemble des partitions possibles.

6.2.1.2 Fonction objectif

La fonction objectif utilisée (équation 6.1) évalue la qualité d'une partition en considérant la cohésion interne des composants ainsi que le niveau de couplage inter-composants. La fonction prend en entrée A, l'ensemble des composants candidats. Elle calcule la moyenne pondérée par la taille de la fonction objectif de chaque composant. Plus le résultat de la fonction objectif est bas, meilleure est la solution. L'évaluation d'un composant dépend principalement de sa cohésion comme on peut le voir dans l'équation 6.2. Cependant, si le couplage dépasse un certain seuil (cm), l'évaluation est lourdement pénalisée. Le seuil cm utilisé correspond au couplage moyen des classes du système, le couplage qu'une classe est calculé avec la métrique CBO (voir chapitre 7). Selon

l'équation 6.2, un composant avec un couplage inférieur à cm aura une évaluation entre $[0, 0.5]$. Au contraire, un composant avec un couplage supérieur à cm aura une évaluation comprise entre $[0.5, 1]$. Le calcul de la cohésion et du couplage sont définis dans les paragraphes suivant.

$$evalArch(A) = \frac{1}{|Cl|} \sum_{C \in A} (evalComp(C) * |C|) \tag{6.1}$$

$$evalComp(C) = \begin{cases} si\ evalCoupling(C) < cm & evalCoh(C)/2 \\ sinon & evalCoh(C)/2 + 0.5 \end{cases} \tag{6.2}$$

Comme nous l'avons vu, un "bon" composant doit inclure toutes les classes qui interagissent pour fournir une fonctionnalité spécifique. Cela est mesuré par sa cohésion interne. Cette mesure (équation 6.3) évalue la distance moyenne entre les classes d'un composant dans les traces d'exécution. Elle calcule la distance moyenne entre les paires de classes appartenant au composant considéré pour toutes les traces ($|C|^2 - |C|$ étant le nombre de paires de classes contenues dans le composant $|C|$). Dans le cas où le composant C contient une classe seule, $evalCoh(C)$ est égale à 1. Cette mesure de cohésion permet d'identifier des ensembles de classes qui ne sont pas en relation directe et évite ainsi le problème rencontré dans la première approche d'identification des composants présentée dans la section 6.1.

$$evalCoh(C) = \frac{1}{|C|^2 - |C|} \sum_{x \in C} \sum_{y \in C, y \neq x} dist(x, y, C) \tag{6.3}$$

La distance entre deux classes a et b dans un composant C ($dist(a, b, C)$ dans équation 6.4) est la moyenne des distances minimales ($distMin(x, y, C)$) entre chaque instance x de la classe a l'instance y de la classe b dans les traces (l'ensemble des instances de la classe a est noté $obj(a)$).

$$dist(a, b, C) = \frac{1}{d|obj(a)|} \sum_{x \in obj(a)} min_{y \in obj(b)}(distMin(x, y, C)) \tag{6.4}$$

La distance maximale entre deux instances de classe est limitée par le paramètre d. Ce paramètre explicite la notion de relation entre instances de classes dans une trace. Je considère que deux instances de classe sont en relation si la distance entre elles est inférieure à d. De plus, ce dernier permet de réduire le temps de calcul en évitant d'explorer trop de chemins pour calculer $distMin(x, b, C)$. Comme la distance entre les instances de classes est limitée par d, j'utilise aussi ce paramètre pour normaliser la distance moyenne $dist(a, b, C)$ entre les classes. S'il n'existe pas de chemin de taille inférieur à d alors $distMin(x, b, C)$ est égal à d. Enfin, un chemin n'est valide que si tous les noeuds qui le composent sont des instances des classes de l'ensemble C.

Une des forces des architectures à base de composants est que les composants sont faiblement couplés et sont composés pour construire des systèmes. La fonction $evalCoupling(C)$ (Equation 6.5), qui est utilisée dans l'équation 6.2, évalue le couplage entre le composant C et les autres composants dans la solution évaluée. Le couplage est mesuré par le nombre de classes à l'extérieur de C qui sont connectées aux classes de C (appelantes ou appelées).

$$evalCoupling(C) = |\bigcup_{c_1 \in C, c_2 \in Cl \setminus C} connect(c_1, c_2)| \tag{6.5}$$

Pour illustrer le calcul de la fonction objectif sur une solution, considérons l'exemple donné par la figure 6.6. Si d est fixé à 3, la solution $S = \{\{A, B, C\}, \{D, F\}, \{E, H, I, J\}\}$ produit les valeurs intermédiaires suivantes :

- $cm = 2.66$
- $evalCoh(\{A, B, C\}, 3) = 0.47$
- $evalCoh(\{D, F\}, 3) = 0.33$
- $evalCoh(\{E, H, I, J\}, 3) = 0.7$
- $evalCoupling(\{A, B, C\}) = 2$
- $evalCoupling(\{D, F\}) = 2$

- $evalCoupling(\{E, H, I, J\}) = 4$

Finalement, le résultat de la fonction objectif pour S est :

$evalArch(S) = \frac{((0.47/2)*3+(0.33/2)*2+((0.7/2+0.5)*4))}{9} = 0.49$

1 **Algorithme :** algorithme génétique généraliste
2 Création d'une population initial P_0,$P_i := P_0$
3 **while** *condition d'arrêt non vérifiée* **do**
4 $eval\ P_i$
5 $P'_i := Selection(P_i)$
6 $P''_i := Croisement(P'_i)$
7 $P_{i+1} := Mutation(P''_i)$
8 **end**
9 retour du meilleur chromosome

Algorithme 3: Algorithme Génétique de base

6.2.1.3 recherche globale : algorithme génétique

Un algorithme génétique est une méta-heuristique globale qui imite le concept d'évolution. L'algorithme commence avec une population initiale (P_0) contenant un ensemble de solutions (appelées "chromosomes") et simule le passage de génération en génération à partir de cette population initiale.

Le fonctionnement général d'un algorithme génétique (Algorithme 3) est le suivant. Pour chaque itération de l'algorithme, une nouvelle population P_{i+1} est produite après avoir appliqués les opérateurs de sélection, croisement puis mutation sur la population courante P_i. Quand le critère d'arrêt est atteint, la meilleure solution trouvée est retournée. Ce schéma général peut être instancié de différente façon en fonction du problème cible.

L'algorithme précis utilisé est décrit dans l'Algorithme 4. Il utilise trois paramètres :

- $MaxIter$ est le nombre maximal de générations pour l'évolution.
- $MaxNiter$ définit le nombre maximal de générations possibles sans nouvelle meilleure solution.
- $SizePop$ borne la taille maximale de la population.

Algorithme : $algoGenetique(MaxIter, MaxNiter, SizePop)$
$iter := 0; niter := 0$
Création d'une population initial P_0de taille $SizePop$
$Best := min_{c \in P_0}(evalArch(c, d))$
while $(iter < MaxIter)\ and\ (niter < MaxNiter)$ **do**
    $eval\ P_{iter}$
    $P'_{iter} := Selection(P_{iter})$
    $P_{iter+1} := \emptyset$
    for $i := 0; i < SizePop; i := i + 2$ **do**
        croisement de c_i c_{i+1} avec la probabilité p_c en c'_i c'_{i+1}
        mutation de c'_i c'_{i+1} avec la probabilité p_m en c''_i c''_{i+1}
        $P_{iter+1} := P_{iter+1} \cup \{c''_i, c''_{i+1}\}$
    end
    $BestLocal = min_{c \in P_{iter+1}} evalArch(c, d)$
    if $evalArch(BestLocal, d) < evalArch(Best, d)$ **then**
        $Best := BestLocal$
        $niter := 0$
    end
    $P_{iter+1} := P_{iter+1} \cup \{Best\}$
    $iter++; niter++$
end
return $Best$

Algorithme 4: Algorithme Génétique

Les paragraphes qui suivent expliquent les choix réalisés lors de l'utilisation de cet algorithme génétique.

Population initiale. Celle-ci est générée de façon aléatoire. Ainsi, j'évite au maximum le risque de converger prématurément vers un optimum local en biaisant les solutions initiales.

Sélection. A partir de la population courante, cet opérateur sélectionne les n chromosomes qui vont servir de base à la population suivante. Trois méthodes de sélection ont été testées :

- La loterie biaisée (roulette wheel) de Goldberg [Gol89] : on sélectionne les n chromosomes un à un. La probabilité $Pr(c, P)$ de sélectionner un chromosome c dans la population courante P dépend de sa qualité $Q(c)$ par rapport aux autres chromosomes de la population. Cette probabilité est donnée par la fonction :

$$Pr(c, P) = \frac{Q(c)}{\sum_{a \in P} Q(a)}$$

- La méthode "élitiste" : seuls les meilleurs chromosomes sont sélectionnés.

- La sélection par tournois : dans la population courante, on choisit aléatoirement $2m$ chromosomes avec $2m < n$. Dans cet ensemble, seul les m meilleurs chromosomes sont sélectionnés. Cette opération est répétée jusqu'à avoir n chromosomes.

Après différent test, la sélection par tournois c'est révélée la meilleure dans ce cas. La méthode élitiste converge prématurément vers un optimum local. Avec la loterie biaisée, le problème est inverse : dans le cas où les chromosomes de la population ont presque la même évaluation, la sélection par cette méthode revient presque à une sélection aléatoire. Ainsi, l'algorithme ne convergeait pas.

De plus, à la fin de la boucle principale de l'algorithme, le meilleur chromosome de la génération précédente est systématiquement ajoutée à la nouvelle génération. Ainsi, l'algorithme devient élitiste.

Croisement. L'opérateur de croisement est binaire, il permet d'obtenir deux nouveaux chromosomes enfants à partir de deux chromosomes parents. L'idée est que les enfants ainsi obtenus ont des chances de bénéficier des bonnes propriétés des deux parents.

Le croisement classique consiste à couper deux chromosomes c_1 et c_2 en deux parties puis à coller la première partie de c_1 (respectivement la deuxième partie de c_1) à la deuxième partie de c_2 (respectivement à la première partie de c_2). Cependant, ce croisement peut produire une solution qui ne satisfait pas les propriétés de complétude et cohérence. En effet, certaines classes peuvent appartenir à plusieurs composants dans les nouveaux chromosomes produits ou, au contraire, ne pas être dans un composant. Pour préserver les deux propriétés mentionnées ci-dessus, la variation suivante [Fal98] est utilisée :

- Division du chromosome c_1 (respectivement c_2) en deux parties c_{11} et c_{12} (respectivement c_{21} et c_{22}), chacune contenant un sous-ensemble du composant.
- Création du chromosome c'_1 en insérant c_{11} entre c_{21} et c_{22} (respectivement c'_2 en insérant c_{21} entre c_{11} et c_{12})).
- Suppression dans c_{21} et c_{22} de toutes les classes présentes dans c_{11} (respectivement dans c_{11} et c_{12} de toutes les classes présentes dans c_{21}).

Par exemple, les chromosomes :
$c_1 = \{\{A, C, I\}, \{E, J\}, \{D, H, B, F\}\}$
$c_2 = \{\{A, H\}, \{B, C, D, E\}, \{F\}, \{I, J\}\}$
sont partitionnés en deux :
$\{\{$**A,C,I**$\}\}$ et $\{\{E, J\}, \{D, H, B, F\}\}$ pour c_1,
$\{\{$**A,H**$\}, \{$**B,C,D,E**$\}\}$ et $\{\{F\}, \{I, J\}\}$ pour c_2.
et produisent les deux chromosomes :
$c'_1 = \{\{\not{A}, H\}, \{B, \not{C}, D, E\}, \{$**A,C,I**$\}, \{F\}, \{I, J\}\}$, et
$c'_2 = \{\{\not{A}, \not{C}, I\}, \{$**A,H**$\}, \{$**B,C,D,E**$\}\{\not{E}, J\}, \{\not{D}, H, \not{B}, F\}\}$. Comme on peut le voir, cet opérateur de croisement préserve les propriétés de cohérence et de complétude.

Mutation. L'opérateur de mutation est unaire (il ne s'applique que sur un chromosome). Il permet de générer un nouveau chromosome légèrement différent de l'original et apporte ainsi de la diversité dans le matériel génétique de la population courante.

L'opérateur de mutation est concrétiser par l'une des trois sous-opérations suivantes :

- *split* : division d'un composant en deux composants.
- *merge* : union de deux composants.
- *move* : déplacement d'une classe d'un composant à un autre.

Le sous-opérateur à appliquer à un composant est choisi aléatoirement. De plus, les trois opérations conservent les propriétés de cohérence et de complétude.

Par exemple, $c = \{\{A, C, D\}, \{E, J\}, \{B, F, H, I\}\}$ peux être muté en :

- $c_{merge} = \{\{A, C, D\}, \{\textbf{B,E,F,H,I,J}\}\ \}$
- $c_{move} = \{\{A, C\}, \{\textbf{D}, E, J\}, \{B, F, H, I\}\}$
- $c_{split} = \{\{A, C, D\}, \{E, J\}, \{\textbf{B,F}\}, \{\textbf{H,I}\}\}$

Critère d'arrêt. Il est double :

- L'algorithme s'arrête après un certain nombre d'itérations : $MaxIter$.
- Si après $MaxNiter$ itérations l'algorithme n'a pas trouvé de nouvelle meilleure solution, l'algorithme s'arrête.

6.2.1.4 recherche local : algorithme recuit simulé

L'algorithme génétique explore différentes solutions dans un large espace de recherche pour produire une solution proche de la solution optimale. Cette solution est alors utilisée par l'algorithme recuit simulé (SA) comme un point de départ pour l'exploration de son voisinage avec pour objectif l'amélioration de cette solution. L'algorithme est présenté dans l'Algorithme 5.

SA ne manipule qu'une unique solution s à la fois, elle est initialisée avec solution la fournie par l'algorithme génétique. Ainsi, à chaque itération de l'algorithme, la solution courante est comparée à un voisin (s_{neigh}) produit par la fonction $Neigh(s)$. Quand s_{neigh} est meilleur que s (mesuré par la fonction objectif $evalArch$), s_{neigh} remplace s. Autrement, cette solution peut être acceptée avec une certaine probabilité ($e^{\frac{-delta}{T_p}}$), paramétrée par la "température" courante T_p dans l'algorithme. Cette température diminue durant la progression de l'algorithme en fonction du coefficient cof. Cet aspect aléatoire est inclus pour éviter de tomber dans un optimum local. Quand la température courante T_p est inférieure à T_{min} un seuil défini par l'utilisateur, l'algorithme s'arrête et la meilleure solution rencontrée ($Best$) est retournée.

Fonction voisin. La fonction voisin ($Neigh(s)$) utilise l'opérateur de mutations de l'algorithme génétique pour produire un voisin.

```
1  Algorithme : RecuitSimule(s, T_p, T_min, iter, cof)
2  Best := s
3  while T_p > T_min do
4      for i = 0; i < iter; i++ do
5          s_neigh := Neigh(a)
6          delta := evalArch(s, d) − evalArch(s_neigh, d)
7          if (delta < 0) ou (random < e^(−delta/T_p)) then
8              s := s_neigh
9          end
10         if evalArch(s_neigh, d) < evalArch(Best, dL) then
11             Best := s_neigh
12         end
13         T_p := cof ∗ T_p
14     end
15 end
16 return Best
```

Algorithme 5: Algorithme *recuit simulé*

Le résultat retourné par l'algorithme SA correspond à l'ensemble des noyaux des composants de l'application cible.

6.2.2 Ajout des classes manquantes

Cette étape correspond à l'étape 4 de la première approche. Elle n'est nécessaire que si le jeux de cas d'utilisations n'est pas complet. Elle a pour but de compléter les noyaux des composants identifier dans l'étape 1. Pour cela la recherche locale SA est réutilisée. En effet, il n'est pas nécessaire d'explorer tout l'espace des solutions étant donné que la solution recherchée est construite à partir de la solution fournie par l'étape 1.

L'espace de recherche est le même que dans l'étape précédente (l'ensemble de toutes les partitions possibles), mais avec une contrainte supplémentaire : les classes assignées

aux noyaux des composants dans l'étape 1 ne peuvent migrer d'un composant à l'autre. La fonction voisin correspond au sous-opérateur *move* de l'opérateur de mutation et ne peut que s'appliquer sur des classes ajoutées dans cette étape. Comme donnée, j'utilise le graphe d'appels statique de l'application cible. En effet, ce dernier contient forcement les classes manquantes. La fonction objectif de cette étape réutilise aussi l'équation 6.1, mais avec une fonction différente pour évaluer un composant (voir Équation 6.6). Cette fonction considère les dépendances statiques pour évaluer la cohésion et le couplage. La cohésion d'un composant C est évaluée par $evalCoh'(C)$, qui correspond aux nombres de méthodes différentes utilisées entre les classes du composant. Le calcul du couplage réutilise la fonction $evalCoupling(C)$ (voir l'équation 6.5).

$$evalComp'(C) = 1 - \frac{evalCoh'(C)}{evalCoh'(C) + evalCoupling(C)} \tag{6.6}$$

La formulation de l'équation 6.6 est quelque peu différente de celle de l'équation 6.2. En effet, son but est de promouvoir les solutions dans lesquelles les noyaux de composant sont complétés avec les classes manquantes plutôt que les solutions où de nouveaux composants sont crées.

A la fin de cette étape, nous obtenons une partition de l'ensemble des classes de l'application. Cette partition respecte les propriétés de cohérence et de complétude.

6.2.3 Etude de cas

Comme pour la première approche d'identification des composants, deux études de cas ont éte réalisées afin d'évaluer la qualité et les défauts de cette approche. La première porte sur l'interprète du langage Logo, la seconde porte sur l'application Lucene. Ces deux études de cas doivent toujours répondre à la question : **les composants identifiés sont-ils corrects ?**

6.2.3.1 Interprète Logo

Les données utilisées pour l'identification sont les mêmes que dans la section 6.1. Cinq composants sont identifiés. Quatre de ces composants sont corrects : ils corres-

pondent aux composants identifiés avec la première approche. Le dernier composant contient deux classes sans relation entre elles : chacune est liée à une fonctionnalité d'un des autres composants. Ce composant n'a pas de sens.

Les résultats de cette étude de cas sont satisfaisants et similaires à ceux obtenu avec la première approche d'identification des composants. Ainsi, selon cette étude de cas, cette approche est de même qualité et n'apporte pas de régression par rapport à la première.

6.2.3.2 Application Lucene

Lucene est un moteur de générique de recherche de information. Cette application contient 221 classes.

25 composants ont été identifiés, 16 bons et 9 mauvais. Quatre des bons composants contiennent clairement des fonctionnalités uniques. Par exemple, un composant fournit la fonctionnalité liée au parseur des requêtes de recherche. Les 12 bons autres composants fournissent une unique fonctionnalité, mais avec des classes manquantes. De plus, certains composant pourraient être fusionnés, car ils fournissent la même fonctionnalité. Par exemple, la fonctionnalité d'indexation est divisée dans 5 composants différents. Enfin, 9 des composants identifiés ne fournissent pas de fonctionnalité clairement définie.

Les résultats obtenus sur cette application sont mitigés. En effet, plus de la moitié des composants identifiés sont corrects. De plus, cette étude montre que cette approche est capable de traiter des applications de taille raisonnable, ce qui n'est pas le cas de la première approche. D'un autre côté, certains composants sont toujours éclatés en sous composant ou n'ont aucun sens.

Afin d'améliorer les résultats, j'ai décidé de prendre en compte de nouveaux critères (granularité d'un composant et nombre de cycles entre composants) dans la fonction objectif. Jusqu'à maintenant, ces critères ont été ignorés par les approches d'identification des composants. Ainsi, le problème d'identification de composants a été reformulé comme un problème multi-critères et simple objectif (les critères sont agrégés dans une fonction d'évaluation unique).

J'ai donc dû développer une nouvelle fonction objectif avec ces quatre critères. Or,

après de nombreux essais, je suis arrivé à la conclusion suivante (voir section 3.7) : il est très difficile d'agréger ces critères de nature différents dans une fonction d'évaluation unique. De plus, selon l'application cible ou l'attente du concepteur, les paramètres de cette fonction varient énormément.

Ainsi, j'ai décidé de développer une nouvelle approche, qui est multi-critères et multi-objectifs. Celle-ci est présentée dans la section suivante.

6.3 Identification multi-objectifs et multi-critères des composants

Comme nous venons de le voir, l'identification des composants d'une application peut être modélisée comme un problème multi-critères et multi-objectifs.

Cette approche (voir figure 6.7) identifie un ensemble de solutions en utilisant un algorithme génétique multi-critères et multi-objectifs : *Nondominated sorting genetic algorithm II* (NSGA-II). Cet algorithme est basé sur le concept d'optimum de Pareto. Comme dans les deux premières approches, le processus d'identification des composants est réalisé en deux étapes : les noyaux des composants des solutions sont identifiés à l'aide de donnée dynamique puis complétés avec des données statiques. La seconde étape nous fournit un ensemble de solutions qui respectent les propriétés de complétude et de cohérence. Les trois critères qui évaluent la qualité d'une solution sont : la cohésion, le couplage, le nombre de cycles. Ces critères sont contraints par la granularité (paramétrable) des composants. Enfin, la dernière étape consiste en la sélection manuelle d'une solution par le concepteur.

6.3.1 Identification des solutions

Les deux premières étapes du processus défini précédemment reposent sur l'algorithme NSGA-II. Cet algorithme génétique utilise le concept optimum de Pareto. Je vais présenter ce concept avant de décrire l'algorithme NSGA-II.

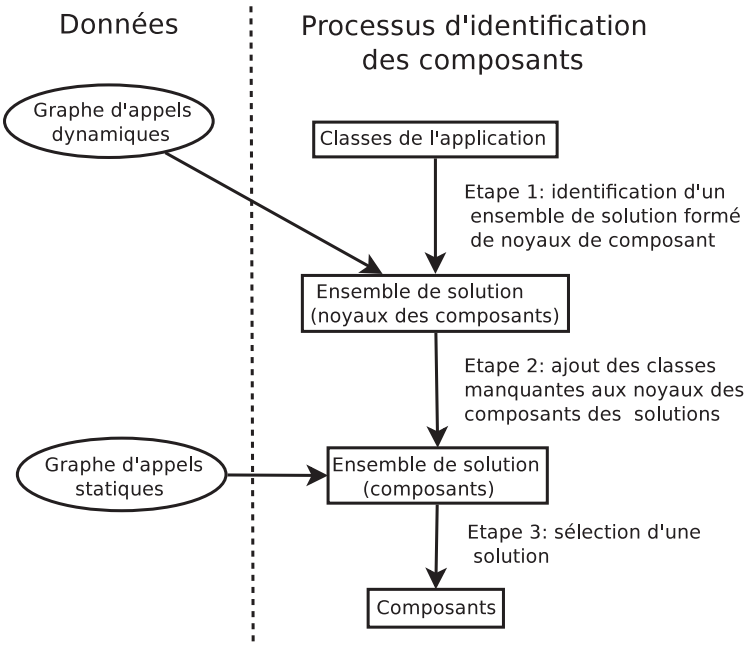


Figure 6.7 – Processus d'identification de composants

6.3.1.1 Concept d'optimum de Pareto.

Dans un problème multi-objectifs, il existe un équilibre entre les différents critères tels que nous ne pouvons pas améliorer un critère sans dégradé aux moins un autre. Cet équilibre est appelé un optimum de Pareto. Une solution est appelée Pareto-optimale si elle n'est pas dominé par une autre solution. La frontière de Pareto est l'ensemble des solutions non dominées. De manière général, si l'on suppose que le but est de maximiser f_i, $i \in 0...N$, la notion de dominance est définie comme suit :

La solution x *domine* x' ssi $\forall i, f_i(x') \leq f_i(x), \exists i, f_i(x') < f_i(x)$

Le concept d'optimum de Pareto permet de respecter l'intégrité de chaque critère car on ne les compare pas entre eux. Une sélection basée sur ce principe va faire converger la population vers un ensemble de solutions efficaces, la frontière de Pareto.

Par exemple, dans la figure 6.8, où chaque solution est représentée par un point, les solutions en blancs sont situées sur la frontière de Pareto et les noires ne sont dominées que par les solutions de la frontière de Pareto.

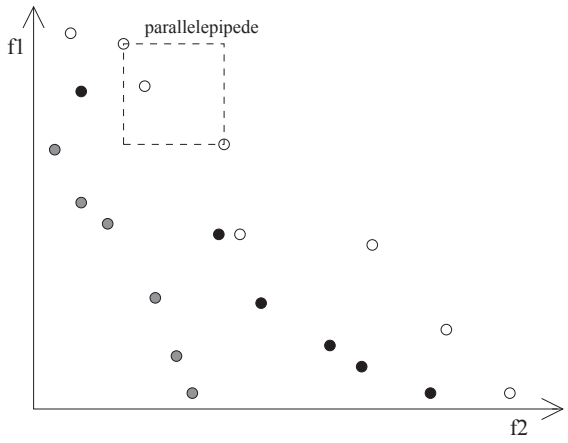


Figure 6.8 – Front de Pareto

6.3.1.2 L'algorithme NSGA-II

Différents algorithmes génétiques multi-critères et multi-objectifs basés sur le principe d'optimum de Pareto ont été proposés dans la littérature [FF93, HHN+94, ZT98, CKO00]. j'ai choisi d'utiliser l'algorithme *Nondominated sorting genetic algorithm II* (NSGA-II) [DPAM02] pour plusieurs raisons : cet algorithme a déjà été validé et utilisé en génie logiciel [LHM07, LHJ10, YH10]. De plus, contrairement aux autres algorithmes multi-critères, l'utilisateur de NSGA-II n'a pas à régler de nouveau paramètre ou à fournir des mesures de distance entre deux solutions. Enfin, NSGA-II est élitiste : je suis sûrs d'avoir en sortie les meilleures solutions rencontrées durant l'exécution de l'algorithme.

1 **Algorithme :** $NSGAII(Generationcounter : t,\ Parentpopulation : P_t,\ Childrenpopulation : Q_t,\ Populationsize : N)$

2 $R_t := P_t \cup Q_t$

3 $\mathcal{F} := fastNondominatedSort(R_t)$

4 $P_{t+1} := \emptyset$ and $i := 0$

5 **repeat**

6 $\quad crowdingFistanceAssignment(\mathcal{F}_i)$

7 $\quad P_{t+1} := P_{t+1} \cup \mathcal{F}_i$

8 $\quad i := i + 1$

9 **until** $\mid P_{t+1} \mid + \mid \mathcal{F}_i \mid \leq N$

10 $Sort(\mathcal{F}_i, \prec_n)$

11 $P_{t+1} := P_{t+1} \cup \mathcal{F}_i[1 : (N - \mid P_{t+1} \mid)]$

12 $Q'_{t+1} := selection(P_{t+1})$

13 $Q''_{t+1} := crossOver(Q'_{t+1})$

14 $Q_{t+1} := mutation(Q''_{t+1})$

Algorithme 6: Boucle principale de l'algorithme NSGA-II

L'originalité de l'algorithme NSGA-II (Algorithme 6), par rapport aux algorithmes génétiques classiques (voir Algorithme 3), réside dans l'étape de sélection qui permet de produire la prochaine génération (lignes 2 à 12). En effet, celle-ci est réalisée en deux étapes. La première étape consiste en une présélection des meilleures solutions (P_{t+1}) dans l'ensemble (R_t) formé par la population courante Q_t et l'ensemble des meilleures solutions déjà rencontré (P_t). Pour cela, NSGA-II utilise un algorithme appelé $fastNondominatedSort$, qui classifie les solutions de R_t selon leur niveau de dominance ($rank$). Les solutions de la frontière Pareto sont assignées au niveau 0 de dominance, les solutions qui sont dominées que par les solutions de niveau 0 sont affectés au niveau 1, etc.. Ensuite, les n meilleurs individus sont pré-sélectionnés (ligne 11) : NSGA-II commence par pré-sélectionner les individus de niveau 0, puis de niveau 1, etc. La pré-sélection s'arrête lorsque l'ensemble est de taille n. Cet ensemble contient les meilleures solutions rencontrées jusqu'à ce point (P_{t+1}).

La seconde étape(ligne 12) permet de construire l'ensemble des solutions sélectionnés (Q'_{t+1}). Cela est fait avec un tournoi binaire. La comparaison entre deux solutions est faite avec l'opérateur $\prec_n$:

$$i \prec_n j \; si \; (i_{rank} < j_{rank}) \; ou \; ((i_{rank} = j_{rank}) \; et \; (i_{distance} > j_{distance}))$$

Cet opérateur sélectionne la solution la moins dominée. Si les deux solutions ont le même niveau de dominance, la solution avec la plus grande *crowding* distance est sélectionnée. La *crowding* distance est une mesure de densité correspondant au périmètre du plus grand parallélépipède qui ne contient que la solution (voir figure 6.8). Ainsi, un individu avec une grande *crowding* distance est situé dans une partie de l'espace des solutions qui est peu peuplée. Pour une plus grande diversité d'individu sur la frontière de Pareto, NSGA-II sélectionne les individus avec une *crowding* distance supérieure lorsque les niveaux de dominances sont les mêmes.

Ensuite les opérateurs de croisement et de mutation sont appliqués aux solutions sélectionnées (ligne 13 et 14).

6.3.1.3 Instanciation de l'algorithme NSGA-II

Comme l'algorithme génétique de base présenté dans la section précédente, l'algorithme NSGA-II peut être instancié de différente façon selon le problème. Étant donné qu'une solution est, comme dans la section précédente, une partition des classes de l'application cible, je réutilise les opérateurs du processus d'identification des composants précédent.

Ainsi, la première étape réutilise l'espace de recherche et les opérateurs définis pour l'étape 1 de la seconde approche d'identification des composants (voir section 6.2)

De même, la seconde étape réutilise l'espace de recherche défini pour l'étape 2 de la seconde approche. l'opérateur de mutation de cette étape correspond à la fonction voisin de l'étape 2.

6.3.1.4 Critères d'évaluation

Contrairement au paradigme objet [CK94, BDW99, BDW97], il n'existe pas de suite standard de métriques pour évaluer la qualité d'une application à base de composants. De plus, les seules métriques développées spécialement pour évaluer de telles applications [CKK01, GG03, LNH07, KS08] sont soit trop vague pour être appliquées à ce problème [GG03], soit inapplicable à ce problème (par exemple Cho et al. [CKK01] propose des métriques pour évaluer l'adaptation et réutilisation des composants), soit déjà couvertes par les notions de cohésion et couplage utilisées dans les deux sections précédentes (par exemple, la métrique *Component Static Complexity* définie dans [CKK01] correspond à ma définition de la cohésion ou la métrique *Component Interaction Density Metric* définie dans [LNH07] à ma définition du couplage). Ainsi, pour évaluer la qualité d'une solution j'ai choisi d'utiliser les trois métriques de qualité suivantes : la cohésion, le couplage et le nombre de cycles. Le nombre de cycles, bien qu'initialement proposé pour évaluer la qualité des packages dans le paradigme objet [Mar], me semble toujours avoir du sens dans le paradigme composant. En effet, les dépendances cycliques entre composants dans une ABC rend celle-ci plus difficile à maintenir et à faire évoluer.

De plus, ces critères sont soumis à une contrainte qui est configurable par le l'utilisateur de l'approche : la granularité des composants.

Cohésion Comme nous l'avons vu dans les deux sections précédentes, un bon composant devrait inclure toutes les classes qui interagissent entre elles pour fournir une fonctionnalités. La force de ces interactions est ce que l'on appelle la cohésion. L'équation 6.7 évalue la cohésion d'une partition de A. Elle correspond à la somme de la cohésion de ses composantes. La cohésion d'un composant correspond au couplage moyen des classes qu'il contient, ce couplage est calculé à partir du nombre de méthodes différentes appelées entre ses classes ($fctCall$).

$$Cohesion(A) = \sum_{C \in A} \frac{\sum_{c_1, c_2 \in C, c_1 \neq c2} |fctCall(c_1, c_2)|}{|C|} \tag{6.7}$$

Couplage Un des critères forts du développement à base de composants est que les composants doivent être faiblement couplés. Le couplage d'une partition A est la somme du couplages de chacun de ses composants (équation 6.8). Le couplage d'un composant est le nombre total de connexions de ses classes avec les classes se trouvant à l'extérieur du composant.

$$Coupling(A) = \sum_{C \in A} | \bigcup_{c_1 \in C, c_2 \in Cl \setminus C} connect(c_1, c_2)| \tag{6.8}$$

Cycle Ce critère correspond au nombre de cycles dans lesquels sont impliqués des éléments de la partition. Ce critère doit être minimisé et le but ultime est de ne pas avoir de cycle.

Taille La taille d'un composant correspond au nombre de classes qu'il contient. Il n'y a pas de consensus sur la taille idéal des composants ou packages. Par exemple, pour Coad et al. [CY91] cette taille doit être comprise entre 5 à 9 classes. D'un autre côté, Lakos [Lak96] exprime cette taille en nombre de lignes de code : entre 500 et 1000 pour un composant de bas niveau et entre 5000 et 50000 pour un composant composite. Enfin, pour Meyer [Mey95] un cluster doit contenir de 40 à 50 classes et être entièrement compréhensible par une seule personne. Ainsi, je n'est défini aucune valeur pour les bornes minimal ($Smin$) et maximal($Smax$) de la taille d'un composant. Celles-ci doivent être fournies par l'utilisateur de l'approche en fonction de la granularité des composants souhaitée. La fonction qui l'évalue ($Size(A) \in [0, 1]$) est la suivante :

$$Size(A) = \frac{1}{|A|} \sum_{C \in A} \begin{cases} \frac{|C|}{Smin} si\ |C| < Smin \\ \\ \frac{Smax}{|C|} si\ |C| > Smax \end{cases} \tag{6.9}$$

Utiliser la taille comme critère peut mener à des résultats incohérents. En effet, ce critère n'a pas le même niveau sémantique que les trois autres et l'utiliser tel quel donne des solutions se trouvant sur la frontière de Pareto, mais incohérentes. Par exemple, une solution dont les composants respectent le critère de taille, mais une cohésion nulle.

Ainsi, ce critère est utilisé comme une contrainte sur le calcul des autres critères : la cohésion, qui doit être maximisée, est multipliée par cette métrique, tandis que le couplage et les cycles, qui doivent être minimisés, sont divisés par cette même métrique.

6.3.2 Sélection manuelle d'une solution

A la fin de la seconde étape, on obtient un nombre raisonnable de partitions de l'application cible (celles sur la frontière de Pareto). Dans cet ensemble de solutions, le concepteur doit choisir une solution. Ce choix peut se faire en fonction de l'importance qu'il accorde à chaque critère qui évalue une solution ou après un examen manuel de chaque solution proposée.

Après la sélection, nous obtenons l'ensemble des composants de l'application cible. Cette partition respecte les propriétés de cohérence et de complétude.

Cette approche est la plus aboutie des trois. En effet, c'est elle qui répond le mieux aux problématiques identifiées dans la section 3.7. Sa validation est présentée dans le chapitre 8. Ainsi, dans la suite de ce mémoire lorsque je parlerai d'identification de composants je me référai à cette méthode.

6.4 Raffinement des composants identifiés

L'étape d'identification des composants fournit automatiquement une partition de l'ensemble des classes de l'application cible. Cependant, la nature même de l'algorithme utilisé dans l'étape précédente, une méta-heuristique, fait que nous ne sommes pas sûrs d'obtenir la solution optimale, mais seulement une solution proche. De plus, comme nous l'avons vu dans la section 3.7, l'application cible peut contenir des *bad smell* qui biaisent le processus d'identification des composants. Ainsi, il est possible qu'une solution soit optimale pour les critères d'évaluation, mais non satisfaisante pour l'expert qui utilise cette approche.

Ainsi, une étape a été ajoutée, appelée raffinement collaboratif des composants, dans laquelle un expert de l'application peut affiner la solution en fonction de certaines infor-

mations sur les composants identifiés. Cette étape n'est pas obligatoire.

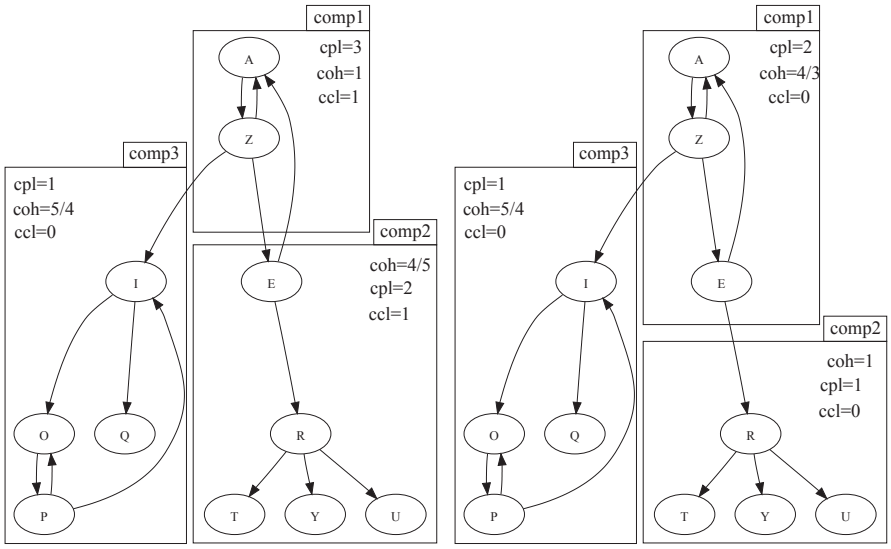
Les informations dont dispose l'expert pour raffiner la solution choisie sont :

- Une représentation graphique des composants et de leur relations.
- Pour chaque composant, son évaluation selon les critères utilisés dans l'étape précédente pour les identifier (couplage, cohésion, cycle,...).
- Pour chaque classe, son évaluation (couplage, cohésion) par rapport à son composant.
- Pour chaque classe, une liste des déplacements possibles entre deux composants. On propose de déplacer une classe d'un composant à un autre si cela améliore l'évaluation d'au moins un critère dans chaque composant.

Après chaque modification effectuée par l'expert sur la solution, les information relative à cette solution sont mises à jour.

6.4.1 Illustration du raffinement

Illustrons ceci par un exemple. Dans cet exemple, chaque appel entre classes est fait avec une unique méthode et une seule fois. Ainsi $nbCall$ et les ensembles $fctCall$ ne figures pas sur la représentation graphique de la solution. Ici le critère de granularité n'est pas pris en compte. La figure 6.9a donne la représentation graphique de la solution ainsi que l'évaluation de ses composants. Le tableau 6.I donne pour chaque classe son couplage (cpl) (nombre de liens avec des classes n'appartenant pas à son composant), sa cohésion (coh) (nombre de liens avec les classes du composant), le nombre de cycles entre composants (ccl) dans lequel la classe est impliquées et enfin le ou les composants ou elle pourrait être déplacée (notation : a($coh \downarrow$) $\rightarrow$ b($cpl \uparrow$) : déplacer la classe entre les composants a et b dégrade la cohésion de a et améliore le couplage de b).



(a) Composant identifiés (b) Composants raffinés

Figure 6.9 – Représentation graphique des composants identifiés

Comme on peut le voir dans le tableau 6.I deux déplacements de classes sont proposés. Après que l'expert est choisi de déplacer la classe E du composant $comp2$ vers le $comp1$, les informations sont mises à jour dans la figure 6.9b et le tableau 6.II. Aucune nouvelle transformation n'est proposée et l'expert n'applique pas de transformation de son propre chef. La figure 6.9b donne les composants finaux de l'application cible.

Après cette étape, nous obtenons les composants finaux de l'application cible. La prochaine étape consiste à projeter ces composants le modèle de composant (voir figure 4.1) pour obtenir l'architecture à base de composants de l'application cible.

6.5 Identification des interfaces

Dans les étapes précédentes, nous avons identifié des groupes de classes qui travaillent ensemble pour fournir des fonctionnalités de haut niveau du système cible : des composants. Néanmoins, pour construire une vision architecturale du système cible avec ses éléments, identifier leur structure interne (les classes et leurs relations qui les forment) n'est pas suffisant. Il faut aussi identifier et définir leur structure externe qui

composant	classe	*cpl*	*coh*	*ccl*	déplacer vers
comp1	A	1	2	1	
comp1	Z	2	2	1	comp1($coh \downarrow$,$cpl \uparrow$) $\rightarrow$ comp2($cpl \downarrow, coh \uparrow$)
comp2	E	1	1	1	comp2($cpl \uparrow, coh \uparrow, ccl \uparrow$) $\rightarrow$ comp1($coh, cpl \uparrow, ccl \uparrow$)
comp2	R	0	4	0	
comp2	T	0	1	0	
comp2	Y	0	1	0	
comp2	U	0	1	0	
comp3	I	1	3	0	
comp3	O	0	3	0	
comp3	Q	0	1	0	
comp3	P	0	3	0	

Tableau 6.I – Information relative à la solution de la figure 6.9a

composant	classe	*cpl*	*coh*	*ccl*	déplacer vers
comp1	A	0	3	1	
comp1	Z	1	3	1	
comp2	E	1	2	1	
comp2	R	0	4	0	
comp2	T	0	1	0	
comp2	Y	0	1	0	
comp2	U	0	1	0	
comp3	I	1	3	0	
comp3	O	0	3	0	
comp3	Q	0	1	0	
comp3	P	0	3	0	

Tableau 6.II – Information relative à la solution de la figure 6.9b

les lie entre eux dans l'architecture : leurs interfaces fournies et requises. Pour définir les interfaces fournies (respectivement requises), il faut d'abord identifier leurs services fournis (respectivement requis) puis les organiser dans des interfaces cohérentes. De plus, ce processus doit produire une ABC conforme au modèle de composants utilisé dans ce processus.

6.5.1 Identification des services

Comme il est spécifié dans le modèle de composants, les services fournis et requis correspondent, respectivement, aux appels de méthodes entrant et sortant du composant et aux accès en lectures/écritures sur les attributs entre composants. L'identification des services est faite à partir d'un graphe d'appels du système cible. Les services requis des

composants sont les méthodes/attributs des classes qui n'appartiennent pas au composant et qui sont utilisés par des classes du composant. De même, les services fournis correspondent aux méthodes/attributs des classes du composant utilisées par des classes extérieures au composant.

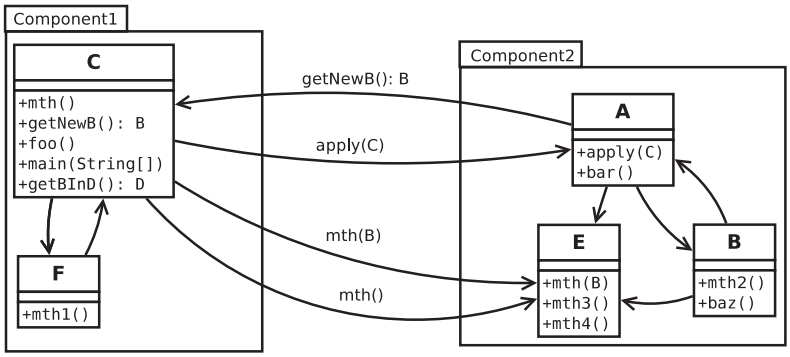
Le graphe d'appels utilisé pour extraire les services doit être à la fois précis et aussi complet que possible. En effet, si le graphe d'appels contient trop d'appels, les interfaces seront bruitées par des services inutiles (jamais utilisés). De plus, une telle situation peut créer des dépendances inutiles entre composants. En revanche, si le graphe d'appels n'est pas complet, des services fournis/requis pourraient être omis, ce qui conduirait à une architecture inexacte de l'application.

Pour ces raisons, le graphe d'appels utilisé pour identifier les services correspond à la fusion des graphe d'appels statique et dynamique utilisés dans l'étape 2 du processus d'identification d'une ABC. L'analyse de ce graphe d'appels permet d'identifier facilement les services requis et fournis de chaque composant. Par exemple, dans la figure 6.10a, l'ensemble des services fournis par `Composant1` est {`C::getNewB():B`}, et l'ensemble de ses services requis est {`A::apply(C)`, `E::mth()`, `E::mth(B)`}.

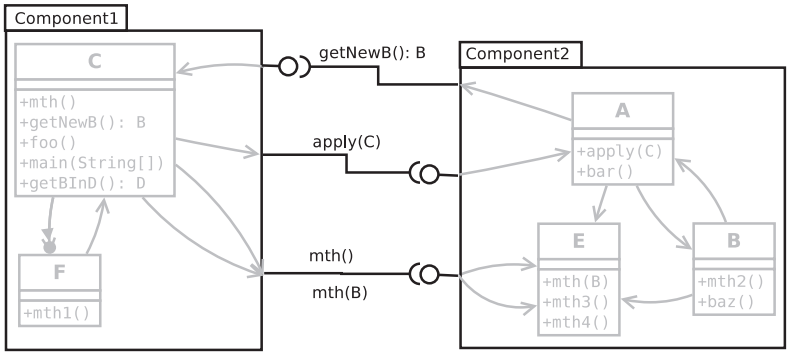
6.5.2 Définition des interfaces des composantes

Jusqu'à présent, seul les services requis et fournis de chaque composant ont été identifiés à l'aide du graphe d'appels. Pour obtenir les interfaces requises et fournies d'un composant, ses services fournis, respectivement requis, doivent être regroupés en sous-ensembles cohérents, selon leur domaine d'application. Comme l'application a été construite en utilisant le paradigme orienté objet, j'utilise ce paradigme pour identifier ces sous-ensembles.

Je commence par identifier les sous-ensembles représentant les interfaces fournies de chaque composant. Pour cela, je regroupe dans la même interface, les services fournis d'un composant qui sont déclarés dans la même classe. Ainsi, un composant aura autant d'interfaces fournies que de classes dont les méthodes sont utilisées depuis l'extérieur du composant. De plus, comme le montre la figure 6.10b, le nombre de services fournis dans une interface peut être inférieur au nombre des méthodes de la classe correspondante.



(a) Graphe d'appels de l'application



(b) Architecture à base de composants de l'application

Figure 6.10 – Identification des interfaces

Par exemple, l'interface fournie de la classe `C`, ne contient que le service `getNewB()`. Les autres méthodes de cette classe sont uniquement utilisées par les classes de son composant (`Composant1`).

Les interfaces requises d'un composant sont construites en analysant ses besoins. Lorsqu'un composant nécessite au moins un service d'un autre composant, une interface requise est construite, de même type que l'interface fournie contenant ce service. Par exemple, le composant `Composant1` de la figure 6.10b a deux interfaces requises, correspondantes aux interfaces fournies par le composant `Component2`.

Finalement, les composants (ensemble de classes) et leurs interfaces (ensemble de méthodes/attributs) donnent une représentation architecturale de l'application cible, basée sur le paradigme composant, et instance du modèle de composant : une architecture à base de composant. Cette ABC respecte les propriétés de complétude et cohérence. Comme nous le verrons dans le chapitre 9, cette ABC peut être utilisée comme modèle pour une réingénierie complète de l'application OO cible en application à base de composants.

Quatrième partie

Validation

Dans cette partie, je présente les travaux réalisés afin de valider le processus d'identification d'une ABC que je propose ainsi que les différents choix faits durant le développement de ce processus.

Dans le chapitre 7 je présente une étude de cas qui a pour but la mise en évidence des problèmes liés au graphe d'appel construit statiquement. Cette étude permet ainsi de valider le choix fait d'utilise des données dynamiques. Cela est réaliste sur le calcul de la métrique de couplage CBO.

Dans le chapitre 8 je présente plusieurs études de cas qui ont pour but d'évaluer l'étape d'identification des composants présente dans la section 6.3.

Enfin, le chapitre 9 montre la faisabilité de restructuration de l'architecture à base de composant identifier vers un modèle concret de composant. Dans ce cas, le framework OSGi.

CHAPITRE 7

IMPACT DES GRAPHES D'APPELS STATIQUES SUR CBO

Pour mettre en évidence les problèmes liés à la construction statique des graphes d'appels, je présente dans ce chapitre une étude qui porte sur le calcul de la métrique de couplage CBO à partir d'un graphe d'appels statiques. Les graphes d'appels sont calculés avec 5 algorithmes différents. Ce travail a été publiés dans [AVDS10].

Dans la section 7.1 la définition formelle de CBO et de son calcul à partir d'un graphe d'appel est donnée. L'étude de cas est présentée dans la section 7.2.

7.1 Métrique de couplage CBO

Chidamber et Kemerer [CK94] ont proposés un ensemble de métriques pour évaluer la qualité des systèmes OO. *Coupling Between Objects* (CBO) fait partie de cet ensemble de métriques. Le but de CBO est de mesurer le nombre de connexions entre une classe donnée et les autres classes du système. Les métriques de couplage définies par Chidamber et Kemerer ont été plus tard formalisées par Briand [BDW99].

Selon la définition originale de CBO, donnée par Chidamber et Kemerer, le CBO d'une classe est le nombre de classes au quelles elle est couplée. Une classe c est dite *couplée* à une classe d si c utilise d ou d utilise c. Une classe c utilise une classe d si une de ses méthodes invoque une méthode définie dans la classe d ou accède à un attribut défini dans la classe d. Plus formellement, Briand définit CBO comme suit :

$$CBO(c) = |d \in C - \{c\} | uses(c,d) \vee uses(d,c)|$$

Où C est un ensemble de classes, $c \in C$, et

$$uses(c,d) = (\exists m \in M_I(c) : \exists m' \in M_I(d) : m' \in PIM(m))$$

$$\vee (\exists m \in M_I(c) : \exists a \in A_I(d) : a \in AR(m))$$

Où $A_I(c)$ est l'ensemble des attributs définis dans la classe c, $AR(m)$ est l'ensemble des attributs référencés dans la méthode m, $M_I(c)$ est l'ensemble des méthodes implémentées dans la classe c et $PIM(m)$ est l'ensemble des méthodes polymorphiquement invoqué par m. Un attribut a appartient à $AR(m)$ si a est lu ou écrit dans le corps de la méthode m.

7.1.1 Calcul de CBO en utilisant un graphe d'appels

A l'exception de PIM, tous les ensembles nécessaires au calcul de CBO peuvent être facilement, déterminés pour chaque classe, avec une simple analyse statique du code source de l'application cible. Le calcul de l'ensemble $PIM(m)$ est plus complexe. Cependant, il peut être aisément calculé à partir d'un graphe d'appels en agrégeant les cibles de la méthode m dans le graphe. Plus formellement, soit un graphe d'appels $CG(M, E)$ où M est un ensemble de méthodes dans un programme et E l'ensemble des arcs entre ces méthodes. $PIM(m)$ est défini comme suit :

$$PIM(m) = \{m' | (m, m') \in E \wedge c, d \in C \wedge m \in M_I(c) \wedge m' \in M_I(d)\}$$

Autrement dit, $PIM(m)$ est l'union de toutes les méthodes ciblées par chaque site d'appels de la méthode m. En utilisant différents algorithmes de construction des graphes d'appels statique, il est possible de calculer différentes versions de CBO.

7.2 Etude de cas

Dans cette étude de cas je m'intéresse au calcul de CBO pour deux applications en utilisant les graphes d'appels obtenus par les différents algorithmes de construction.

7.2.1 Cadre expérimental

7.2.1.1 Applications

ArgoUML 0.18.1 est un outil de création de diagrammes UML. L'ensemble des classes et interfaces utilisé pour calculer CBO correspond aux éléments du paquetage `org.argouml`. L'ensemble C contient 1237 classes et 100 interfaces.

Azureus 2.1.0.0 est un client BitTorrent multiplateforme qui permet de partager des fichiers sur un réseau pair-à-pair. Cette application contient 1232 classes et 250 interfaces dans le paquetage `org.gudy.azureus2`.

7.2.1.2 Implémentation

L'outil de calcul de la métrique est implémenté en utilisant SOOT, un framework d'analyse statique pour Java [VRGH+00]. SOOT fournit des algorithmes de construction de graphes d'appels présentés dans la sous-section 5.2. Il implémente aussi le chargement dynamique de classes en demandant à l'utilisateur de spécifier l'ensemble des classes qui peuvent être chargées durant l'exécution.

J'ai étendu SOOT pour calculer le CBO à partir d'un graphe d'appels. J'ai aussi ajouté une implémentation de l'algorithme simple de construction des graphes d'appels *Declaring Target*. L'algorithme considère seulement les cibles déclarées dans les sites d'appels et ignore le polymorphisme. Cette méthode est utilisée par tous les outils de calcul de métriques.

Pour le calcul de CBO, je considère seulement le couplage entre les classes d'application. Celui-ci est calculé en utilisant quatre algorithmes de construction de graphe d'appels : *Declaring Target* (DT), *Class Hierarchy Analysis* (CHA), *Rapid Type Analysis* (RTA) et *Variable Type Analysis* (VTA). Dans le cas de VTA, j'ai utilisé deux versions différentes de l'algorithme : une avec le chargement dynamique de classes (noté "VTAd") et une autre sans. Le chargement dynamique de classes n'affecte pas les graphes d'appels construits en utilisant DT et CHA.

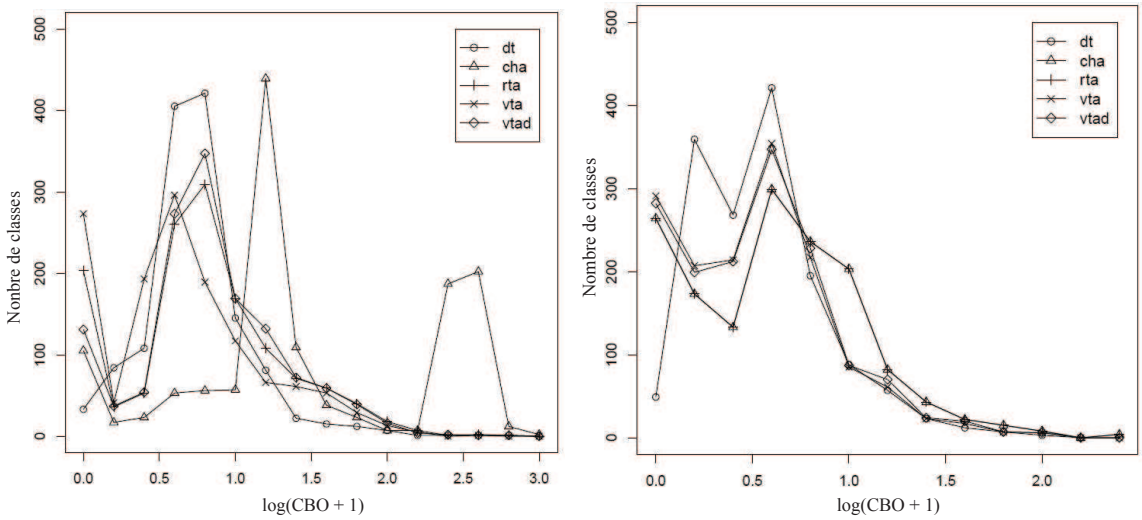
7.2.2 Questions de l'étude

Afin de bien comprendre les différences entres les algorithmes de construction de graphe d'appels sur le calcul de CBO, les résultats sont analysés selon différents points de vu :

- **Distribution des valeurs :** en fonction des algorithmes utilisés, comment sont distribuées les valeurs de CBO ?
- **Code mort :** quel est le comportement des différents algorithmes vis à vis de la détection du code mort (CBO=0) ?
- **Interface :** en fonction de l'algorithme utilisé, quel rôle jouent les interfaces (au sens Java) dans le calcul de CBO ?
- **Polymorphisme** : selon l'algorithme utilisé, quel est l'influence du nombre d'appels entrant ou sortant d'une classe sur le calcul de CBO ?
- **Chargement dynamique de classe :** dans le cas des algorithmes VTA et VTAd, la gestion du chargement dynamique influence-t-elle le calcul de CBO ?

7.2.3 Distribution des valeurs de CBO

Pour étudier l'impact du polymorphisme et du chargement dynamique de classes sur le calcul de CBO, j'ai calculé la distribution des valeurs de CBO en fonction du nombre de classes. La Figure 7.1 montre les distributions pour tous les graphes d'appels construits avec les différents algorithmes. Par ailleurs, dans la figure j'ai utilisé une fonction logarithmique pour la mise à l'échelle sur l'axe horizontal. Les résultats montrent que le choix de l'algorithme a un impact significatif sur des valeurs de CBO pour ArgoUML. Cela indique qu'ArgoUML utilise le polymorphisme de façon non triviale. Les résultats montrent aussi que l'algorithme DT sous-estime la valeur de CBO pour un grand nombre de classes. Cela doit sûrement se produire dans le cas où les sites d'appel sont polymorphes (un site appel avec au moins deux cibles). Pour confirmer cette hypothèse, j'ai examiné les différences de valeurs calculées avec DT et VTAd. Pour la



(a) ArgoUML (b) Azureus

Figure 7.1 – La distribution de CBO

plupart des classes dont la valeur de CBO diffère entre DT et VTAd, VTAd obtient une valeur de CBO plus haute. De plus, même si DT et VTAd aboutissent à la même valeur, l'ensemble des classes couplées n'est pas nécessairement identique. D'autre part, il y a des cas où DT sur-estime le couplage pour une classe. Par exemple, dans ArgoUML, la classe `UmlDiagramRenderer` est couplée à 63 autres classes selon DT, mais seulement à 6 classes selon VTAd. `UmlDiagramRenderer` est une classe abstraite qui fournit les implémentations de deux méthodes publiques, ce qui augmente le couplage avec les classes d'application dans le cas de DT. Les deux méthodes publiques sont redéfinies dans les sous-classes de `UmlDiagramRender` et ne peuvent donc pas être appelées en pratique. VTAd identifie correctement cette situation et calcule un CBO plus bas que DT. En fait, même CHA (qui est moins précis que VTA) est suffisant pour obtenir le même résultat dans ce cas.

La Figure 7.1b montre qu'il y a beaucoup moins de différences entre les divers algorithmes dans le cas d'Azureus. En effet, Azureus utilise très peu d'héritage et ainsi il y a très peu de sites d'appels vraiment polymorphes. L'algorithme de DT, cependant, diffère sensiblement par rapport aux autres algorithmes. En effet, pour DT 49 classes sont inaccessibles (CBO=0) alors que les autres algorithmes il y a entre 264 et 291 classes

inaccessibles. Ceci constitue, en moyenne une surévaluation de la valeur de CBO de 4, et conduit donc aux sommets que l'on voit sur la figure 7.1b autour de 0,2 et 0,6.

7.2.4 Code mort

Un des principaux avantages des graphes d'appels pour le calcul du couplage est sa capacité de détection du *code mort* (le code qui ne peut probablement pas être exécuté pour un programme donné). La quantité de code mort détecté varie d'un algorithme à un autre. Des algorithmes conservateurs comme CHA et VTAd peuvent surestimer la quantité de code mort : ils peuvent estimer qu'une classe est *vivante* alors qu'elle n'est pas utilisée en pratique. Au contraire, l'algorithme DT peut sous-estimer la quantité de code mort dans un programme. Avec DT, un appel virtuel vers une cible déclarée dans une interface couple la classe source avec l'interface, alors que durant l'exécution tous les appels invoqueront obligatoirement des méthodes implémentées dans les sous-classes de l'interface. D'autre part, les surestimations peuvent être causées par la situation inverse. Par exemple, la classe `ModeCreateLink` utilise seulement des méthodes de la bibliothèque (elle n'est donc pas couplée avec les classes d'application) et n'est jamais utilisée comme cible déclarée d'un site d'appel. Donc, pour DT la classe `ModeCreateLink` semble être morte, mais VTAd identifie que les méthodes de cette classe sont accessibles par appels polymorphes. Donc, pour DT la classe `ModeCreateLink` semble être morte, mais VTAd identifie que les méthodes de cette classe sont accessibles par appels polymorphes. La Figure 7.1a montre clairement que la quantité de code mort varie selon l'algorithme de construction de graphe d'appels utilisé. Les classes mortes correspondent aux classes dont la valeur de CBO est 0. La nature très conservatrice de l'algorithme CHA le rend incapable d'identifier la plupart du code mort dans ArgoUML. DT identifie 13 classes comme mortes (CBO=0), mais certaines de ces classes sont des faux positifs (ces classes apparaissent dans des traces d'exécution de l'application). VTAd identifie 32 classes mortes. Les 174 classes identifiées comme morte par VTA ne sont pas vraiment mortes. La plupart de ces classes sont seulement accessibles par chargement dynamique de classes et par réflexion. Par exemple, VTA identifie les 11 sous-classes concrètes de la classe abstraite `Wizard` comme morte alors qu'elles sont

toutes instanciées exclusivement à l'aide de chargement dynamique.

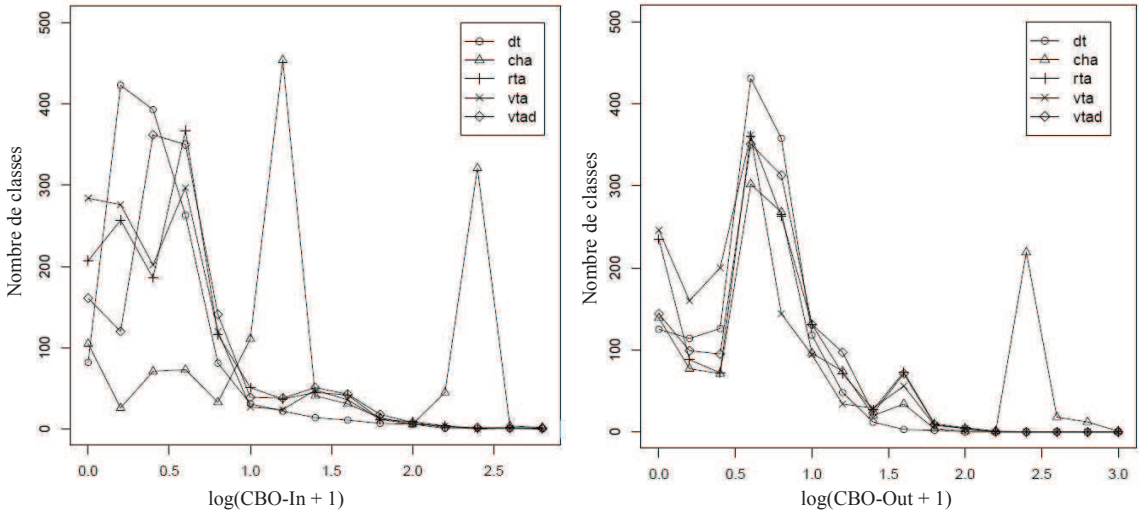
La figure 7.1b montre une tendance similaire pour le code mort dans Azureus. Comme Azureus a plus d'appels résolus statiquement qu'ArgoUML, l'écart entre la quantité de code morte identifiée par les différents algorithmes est beaucoup plus faible que pour ArgoUML. Par exemple, VTAd identifie 282 classes mortes, contre 291 classes pour VTA, 279 pour RTA et 264 pour CHA. Seul DT diffère sensiblement des autres algorithmes : il identifie 49 classes mortes.

7.2.5 Interfaces

Dans le cas de Java, une interface est considérée comme une classe complètement abstraite et ne devrait donc jamais intervenir dans le calcul de CBO. Or, de la même façon que le code mort, l'importance des interfaces dans les calculs de CBO varie énormément selon l'algorithme de construction de graphes d'appels utilisé. Dans un code, il est possible d'utiliser des méthodes d'interfaces comme cible déclarée dans un site d'appel. L'algorithme DT considère ces appels dans les calculs de CBO. Dans le cas des autres algorithmes, les appels polymorphes vers des interfaces sont résolus pour construire le graphe d'appels, et donc ces algorithmes n'incluront pas d'interfaces dans leur calcul de CBO. Dans quelques cas, cela peut mener à des variations extrêmes de CBO. Par exemple, la classe `NSUMLModelFacade` a une valeur de 4 pour CBO avec l'algorithme DT, cette classe n'est jamais appelée directement en utilisant son type statique, mais invoquée à travers l'interface `Facade`. En conséquence, sa valeur de CBO calculée avec VTAd est de 576.

7.2.6 Polymorphisme

Les différents algorithmes de construction de graphes d'appels se distinguent dans leur façon de mesurer CBO, d'une classe C, de deux façons : ils peuvent faire varier l'ensemble de classes utilisées pour les appels *entrant* (appels utilisant des méthodes de C) et pour des appels *sortants* (les méthodes de C qui appellent des méthodes d'autres classes). Dans le but d'obtenir une meilleure compréhension de l'impact des différents



(a) Appels entrant (CBO-In) (b) Appels sortant (CBO-Out)

Figure 7.2 – Appels Entrant/sortant de CBO pour ArgoUML

algorithmes sur le calcul de CBO, le couplage des appels entrants (CBO-In) et des appels sortants (CBO-Out) a été mesuré. La figure 7.2 donne ces résultats. Il est clair que l'algorithme DT sous-estime le couplage pour CBO-In et CBO-Out par rapport à VTAd. Plus intéressant, la figure 7.2a montre que les sur-estimations de CHA proviennent principalement de CBO-In plutôt que de CBO-Out. Dans ce cas, il est possible que certaines classes puisent être appelées depuis plusieurs sites d'appels du programme en fonction de leur hiérarchie de classe, mais en réalité ces appels sont concentrés dans un petit nombre de classes spécifiques plutôt que distribuées à travers toutes les cibles potentielles. Par exemple, la classe `ActionAddMessagePredecessor` hérite de `UMLAction`. Tous les appels qui utilisent la classe générique `UMLAction` comme une cible contribuerons au CBO-In de `ActionAddMessagePredecessor` avec l'algorithme CHA, aboutissant ainsi à un total de 252 pour CBO-In. D'autre part, VTAd détermine que très peu de ces appels ciblent en réalité `ActionAddMessagePredecessor` et obtient donc un CBO-In de seulement 3. Dans les deux cas, CBO-Out pour `ActionAddMessagePredecessor` est de 7.

Les mêmes effets ont été observés sur Azureus, mais les différences sont plus marginales. Cela est dû au fait que la plupart des appels dans cette application ne sont pas

polymorphiques et peuvent être résolus statiquement.

7.2.7 Chargement dynamique de classes

Le chargement dynamique de classes est un des mécanismes de réflexion supportés par les langages modernes, telles que Java, et qui gagne rapidement en popularité. Le tableau 7.I montre l'utilisation de ces mécanismes dans ArgoUML et Azureus. Les résultats de mon investigation montrent que les deux applications font une utilisation non triviale de la réflexion et plus particulièrement du chargement dynamique de classes pour utiliser des classes supplémentaires durant l'exécution (colonne `forName`).

La comparaison du CBO estimé par VTA et VTAd dans la figure 7.1 montre que la différence due au chargement dynamique est très significative. Ne pas représenter de telles caractéristiques déforme artificiellement la distribution de valeurs de CBO et rend leurs utilisations difficiles. Cette différence est due à l'architecture "pluggable" de ArgoUML. Dans ArgoUML, le paquetage `org.argouml.uml.cognitive.critics` fournit cette architecture. Par conséquent, un nombre important de classes ne se trouvent pas dans le graphe d'appels quand la réflexion n'est pas prise en compte.

Pour Azureus, les distributions de CBO calculées par VTA et VTADd sont pratiquement indiscernables bien que l'application utilise la réflexion dans le code source. Par conséquence, je pense que le chargement dynamique de classe ne joue pas de rôle majeur pour Azureus. En fait, une inspection du code source révèle qu'Azureus charge dynamiquement souvent une classe prédéterminée en utilisant une chaine de caractère constante qui contient son nom.

Tableau 7.I – Utilisation du chargement dynamique de classes

	`forName`	`newInstance`	`invoke`
ArgoUML	14	23	6
Azureus	6	7	1

7.3 Conclusion

Cette étude de cas met en évidence que la méthode utilisée pour construire le graphe d'appels à une influence non négligeables sur le calcul de CBO. Par exemple, DT, la méthode utilisée par la plus part des outils de calcul de métrique, pose de nombreux problèmes. Elle crée des liens inexistants entre méthodes ou encore ignore des liens existants. Un des effets de cela est la sous-estimation de l'ensemble des classes mortes avec l'algorithme DT. Un autre exemple de problèmes est le fait que l'algorithme CHA, et même VTAd sous-estiment la quantité de classes mortes. Enfin, dans le cas où un algorithme d'analyse de type plus performant est utilisé, des problèmes apparaissent avec la gestion (ou non) du chargement dynamique. Celle-ci n'est pas triviale et influence le calcul de CBO.

Ainsi, cette étude de cas confirme les problèmes liés à la construction statique des graphes d'appels, problème que j'ai soulevé dans la section 3.7. C'est ainsi que j'ai fait le choix de construire le graphe d'appels avec des données dynamiques (appels détectés lors de l'exécution de l'application).

CHAPITRE 8

IDENTIFICATION MULTI-OBJECTIFS DES COMPOSANTS

Dans ce chapitre, je présente une étude de cas réalisée afin de valider l'étape d'identification des composants. De plus, cette étude de cas doit aussi déterminer si les données dynamiques donnent réellement de meilleurs résultats que les données statistiques. cette étude de cas a été réalisée sur quatre applications : un interprète Logo, COCOME [HKW+08], JEval et PDFsam.

8.1 Cadre expérimental

8.1.1 Applications

L'évaluation de l'étape d'identification des composants est effectuée sur quatre applications :

8.1.1.1 Interprète Logo

Logo est un langage de programmation créé pour l'apprentissage de la programmation et possède pour cela des instructions graphiques très faciles à utiliser. L'interprète a une interface graphique qui permet d'écrire le programme et une fenêtre qui affiche le résultat des instructions graphiques. Logo a été choisi pour deux raisons : i) sa taille (40 classes et 2 interfaces), qui me permet d'effectuer une analyse approfondie des résultats, ii) j'ai participé à son développement, ce qui me permet de proposer une ABC référence. Cette ABC contient 4 composants : le parseur du langage, l'interprète et sa bibliothèque de fonction, la partie graphique du langage et enfin l'interface utilisateur de l'application. L'interprète Logo contient 40 classes.

8.1.1.2 JEval 0.9.4

JEval est une bibliothèque pour l'évaluation d'expressions booléennes, de fonctions mathématiques et d'opérations sur les chaines de caractères. JEval permet l'évaluation des expressions et fonctions statiquement ou dynamiquement durant l'exécution de l'application. JEval est fournie avec des exemples couvrant toutes les fonctionnalités de la bibliothèque ainsi qu'un jeu de tests unitaires complet. Ceci a contribué dans le choix de cette application. En effet, je dispose ainsi d'un ensemble de cas d'utilisations couvrant au maximum l'application. JEval contient 83 classes. Les composants référence ont été identifier après une inspection manuelle de l'application. Ils sont au nombre de cinq et correspondent au parseur/évaluateur ainsi que les différentes bibliothèques de JEval.

8.1.1.3 CoCoME

CoCoME est une application démonstration d'un système commercial distribués. Son architecture est orientée composants et implémentée en Java. L'objectif de ses concepteurs est de montrer comment organiser les packages Java en utilisant le paradigme composant. L'architecture conceptuelle est bien documentée par des cas d'utilisation, diagrammes de composants et des diagrammes de séquences. Ainsi, nous avons une vraie application OO avec les packages correspondant à des composantes. La solution référence de CoCoME contient 7 composants. Ainsi, CoCoME constitue un très bon candidat pour l'évaluation de mon approche. CoCoME contient 127 classes.

8.1.1.4 PDFsam 2.2.1

PDFsam est un outil qui permet la fusion, le découpage, ou bien encore la rotation des pages d'un document pdf. Cet outil fournit une interface graphique qui permet d'effectuer et de visualiser ces opérations. Il peut être aussi utilisé en ligne de commande et étendu via des plugings. PDFsam permet de tester la montée en échelle du processus d'identification des composants. En effet, PDFsam contient 272 classes. Les composants référence ont été identifier après une inspection manuelle de l'application. Ils sont au nombre de 16 et ont une grande variété de granularité (entre 2 et 57 classes).

8.1.2 Implémentation

Tous les outils nécessaires à mon processus d'identification des composants ont été implémentés en Java à l'aide de SOOT [VRGH+00] comme API. SOOT est un framework pour l'analyse statique et la manipulation du Bytecode Java. Il fournit les algorithmes de construction graphes d'appels présentés dans la section 5.2. Les outils développés sont :

- **Traceur :** cet outil permet la génération des traces d'exécution. A l'aide de SOOT, le bytecode des classes cibles est instrumentalisé afin de tracer les appels de méthodes. Par la suite, l'exécution de cas d'utilisation génère des traces sous la forme désirée.
- **IdentComp :** cet outil identifie un ensemble de solutions dans l'application cible (étape de 2 de la figure 1.1). Il utilise les traces d'exécution, obtenue avec le traceur, et éventuellement un graphe d'appels statiques pour produire un ensemble de solutions (composants abstraits).

Pour cette étude de cas, j'ai appliqué une contrainte de taille faible, l'objectif étant d'obtenir un maximum de solutions possibles. Ainsi, j'ai mis la limite inférieure de la taille d'un composant à 1 et la limite supérieure correspond au 1/3 du nombre total de classes. Le graphe d'appels statique a été construit avec l'algorithme VTA. Le tableau 8.1.2 donne le nombre de classes contenues dans les graphes d'appels statiques et dynamiques ainsi que le nombre total de classes contenues dans l'application. Comme on peut le voir, les cas d'utilisation couvrent la majorité des classes des quatre applications. La faible couverture des classes par le graphe d'appels statique de PDFsam peut-être

	dynamique	statique	nb. total de classes
Logo	38	40	40
CoCoME	110	121	127
PDFsam	251	226	272
JEval	83	83	83

Tableau 8.I – Taille des graphes d'appels

expliquée par son utilisation de la réflexivité. Or l'algorithme VTA ne prend pas cela en compte.

8.2 Questions de l'étude

Cette étude de cas doit répondre à trois questions afin d'évaluer l'étape d'identification des composants.

Q1 : Les solutions de référence ne se trouvent pas parmi les solutions proposées ? je possède une décomposition référence pour chacune des applications et je suppose que chaque décomposition référence est de qualité. Si mon approche est valide, cette dernière doit donc appartenir à l'ensemble des solutions proposé, ou du moins proche à une ou plusieurs des solutions de cet ensemble.

Pour évaluer la qualité d'une décomposition par rapport à la référence, j'utilise la métrique de similitude entres partitions : MoJoFM [WT04]. Cette métrique a été spécialement développée pour les décompositions de logiciel. Elle est basée sur la mesure de distance entre partition MoJo [WT03]. Pour MoJo, la distance entre deux partitions correspond au nombre minimal d'opérations (déplacement d'un élément entre deux sous-ensembles ou fusion de deux sous ensemble) pour obtenir deux partitions semblables. MoJoFM normalise cette distance pour la rendre indépendante de la taille des partitions : 100% si les deux partitions sont égales 0% si elles sont complètement différentes. Ainsi, MoJoFM correspond à une métrique de similitude.

J'applique cette métrique entre la solution référence et les solutions fournies par mon approche, puis j'examine en détaille les cinq solutions les plus proches.

Q2 : Y a t-il d'autres bonnes solutions ? Un des choix faits dans mon approche est de fournir un ensemble de solutions. Ce choix est soutenu par l'affirmation qu'il peut y avoir des solutions multiples et seuls les concepteurs sont en mesure de choisir celle qui répond le mieux à leurs besoins. Ainsi, je dois montrer que cette approche fournir plusieurs bonnes solutions.

Pour cela, j'examine en détail 20% des solutions sélectionnées aléatoirement. Ces solutions sélectionnées sont ensuite classées en trois catégories : **BON** si les composants sont parfaits ou ne demandent que très peu de correction manuelle ; **MOYEN** si plus de la moitié des composants sont corrects et permettent ainsi une compréhension partielle de l'application ;**MAUVAIS** si ABC proposée n'a pas de sens.

Pour répondre à cette question, je ne considère que l'interprète Logo. En effet, je ne connais pas suffisamment bien l'architecture des trois autres applications pour juger a posteriori de la qualité d'une décomposition de ces applications.

Q3 : Les données dynamiques donnent-t-elles de meilleurs résultats ? Par rapport aux approches existantes d'identification des composants, une des différences majeures de mon approche réside dans le fait d'utiliser des données dynamiques (les traces d'exécution). J'ai déjà expliqué en quoi cela est préférable à l'utilisation des données statiques (section 3.7 et chapitre 7), mais je dois le montrer sur des applications concrètes.

Pour cela, j'applique les questions **Q1**, **Q2** au résultat obtenu à partir des données statistiques et je compare ces résultats avec les résultats obtenues à partir du graphe d'appels dynamique.

8.3 Résultats

Le tableau 8.2 donne le nombre de solutions obtenues pour les quatre applications avec, respectivement, des données dynamiques et statiques. Le tableau 8.III présente les résultats de similarité (MoJoFM) pour les 5 meilleures solutions avec leur solution de référence correspondante. Le tableau 8.IV donne les résultats de la mesure de qualité

	dynamique	statique
Logo	67	124
CoCoME	132	138
PDFsam	133	69
JEval	75	80

Tableau 8.II – Nombre de solutions

Logo		CoCoME		JEval		PDFsam	
Dynamique	Statique	Dynamique	Statique	Dynamique	Statique	Dynamique	Statique
89.19%	71.05%	81.12%	73.61%	83.33%	81.82%	84.22%	60.66%
86.49%	71.05%	81.12%	70.42%	83.33%	80.00%	83.79%	59.02%
83.78%	68.42%	73.58%	69.44%	81.48%	80.00%	83.35%	56.56%
83.78%	65.78%	73.58%	69.44%	81.48%	79.82%	82.92%	54.92%
81.08%	63.16%	72.64%	68.06%	79.44%	77.36%	82.49%	53.72%

Tableau 8.III – Résultats

pour 20% des solutions prises aléatoirement dans des l'ensemble solutions obtenues pour l'interprète Logo.

8.3.1 Réponse à la question Q1

Comme nous pouvons le voir dans le tableau 8.III, pour toutes les applications la meilleure des solutions ne correspond pas à la référence. Ainsi, aucune des solutions références n'appartient aux solutions proposées.

De ce fait, j'ai examiné plus en détail les cinq meilleurs résultats. Pour cela, je doit d'abord faire correspondre les composants d'une solution avec les composants de la référence. Pour déterminer si deux composants correspondent, j'utilise la relation d'affinité définie par Koschke et al. dans [KE00]. Deux composants a et b correspond ssi :

$$\frac{a \cap b}{a \cup b} > p$$

p est fixé à 0.7 comme le recommande Koschke et al. Avec cette valeur et la propriété de cohérence, il est impossible qu'un composant de la référence corresponde à plus d'un composant de la solution. Quand un composant de la solution n'a pas de composant correspondant dans la référence, je considère que toutes ses classes sont mal placées.

Regardons les résultats pour chaque application.

Interprète Logo Les 5 solutions sont globalement semblables. Dans la meilleure solution, seules 2 classes sur 40 sont mal placées. Globalement, on obtient les résultats suivants : dans les cinq meilleures solutions, sept classes sont mal placées au moins une fois, trois classes sont mal placées 2 fois et 4 classes plus de 4 fois.

Le mauvais placement d'une classe peut être expliqué par deux phénomènes différents :

- Lorsque dans un composant de la solution de référence il existe un sous-ensemble de classes hautement cohésif, mon approche a la tendance à les regrouper dans un composant distinct. Cela augmente la cohésion globale de la solution. Ainsi, au lieu d'obtenir un unique composant, le composant est éclaté dans plusieurs composants. Ces composants doivent être regroupés en un unique composant pour avoir du sens.

- Lorsqu'une classe a un très faible nombre d'appels entrants et aucun des appel sortant. Une telle classe n'a aucune influence sur le nombre de cycles et très peu sur le couplage et la cohésion de son composant. Ainsi, elle peut être située dans n'importe quel autre composant sans changement significatif sur le résultat global.

COCOME Comme nous pouvons le voir les résultats de CoCoME sont de moindre qualité que ceux obtenus avec Logo. Sur les cinq solutions, seuls 3 composants sont correctement retrouvés et peuvent être considérés comme bons. Ces résultats peuvent être expliqués par l'observation suivante : sur les trois critères, seul le nombre de cycles est minimisé dans l'architecture témoin. Mais même pour le même nombre de cycles, mon approche trouve de meilleures solutions. Par exemple, l'ABC référence a 7 cycles, un couplage de 218 et une cohésion de 19,7. Pour une solution avec le même nombre de cycles, mon approche propose une ABC avec un couplage de 102 et une cohésion de 23.5, ce qui est bien meilleur dans les deux cas. En outre, mon approche trouve de meilleures solutions avec un nombre inférieur de cycles (6).

Ne connaissant pas la sémantique des classes de CoCoME, je ne peux pas affirmer que les solutions avec un bon score correspondent à de véritables bonnes solutions. Cependant, les meilleures solutions, qui sont près de la solution de référence, mériteraient un examen par un concepteur de CoCoME.

JEval Les résultats de JEval sont légèrement meilleurs que les résultats COCOME et se dégradent moins rapidement. Sur les cinq composants, les trois qui contiennent le

plus de classe sont toujours retrouvés dans les meilleures solutions à deux ou trois classes près. Ces composants correspondent aux composants ”librairie mathématique”, ”librairie d'opérations sur les chaines de caractères” et enfin le composant lié à l'évaluation d'expression booléenne. Les deux composants restants sont identifiés, mais partiellement. Entre un tiers et la moitié de leurs classes sont mélangées avec les autres composants. Cela s'explique par le fait que ces deux composants sont liés à l'évaluation des fonctions fournies par les trois autres composants ou l'évaluation des fonctions construites à partir de celles-ci. Ils sont ainsi fortement couplés avec ces trois composants.

PDFsam Les principaux composants sont identifiés dans les cinq meilleures solutions. Par contre, les classes des deux composants utilitaires sont éparpillées parmi les composants principaux. De même, les cinq composants qui correspondent au pluging de l'application sont soit fusionnés entre eux, soit fusionnés avec le composant qui les gère. En effet, ces composants ont une granularité très fine (entre 2 et 6 classes) et sont fortement couplés avec le composant qui les gère.

8.3.2 Réponse à la question Q2

Le tableau 8.IV donne l'évaluation manuelle de la qualité de 20% des solutions, pour l'interprète Logo sélectionnées aléatoirement. Parmi ces solutions sélectionnées aléatoirement quatre sont considérées comme bonnes. Parmi ces dernière, une est dans le top cinq (voir tableau 8.III) et une est très différente de la solution de référence. Dans cette dernière, les composants parseur et évaluateur de la solution de référence sont fusionnés en un unique composant de même que les composants correspondant à l'in-

	Logo	
	Dynamique	Statique
BON	4	0
MOYEN	7	16
MAUVAIS	3	8
Total	14 sur 67	24 sur 124

Tableau 8.IV – Qualité des solutions

terface utilisateur (GUI) et l'affichage graphique (Display). Ainsi, nous obtenons une solution faite de deux composantes : le premier (Évaluateur + Parser) encapsule la fonction d'interprétation du langage Logo, tandis que le second (GUI + Display) offre les fonctionnalités liées à l'interaction de l'utilisateur et l'affichage des résultats de cette interaction. Cette solution est tout à fait correcte.

Ainsi, cette approche propose plusieurs solutions acceptables avec des valeurs légèrement différentes pour les critères d'évaluation. Il est donc intéressant de proposer au concepteur plusieurs solutions pour qu'il choisisse celle qui répond le mieux à ses besoins.

8.3.3 Réponse à la question Q3

Les résultats des question **Q1** appliquées aux quatre applications avec un graphe d'appels statique se trouvent dans les tableaux 8.III. Ceux-ci montrent clairement que les graphes d'appels dynamiques donne de meilleurs résultats que les graphes d'appels statiques.

Dans le cas des application PDFsam et de l'interprète Logo, cela peut être expliqué par leur utilisation intensive du chargement dynamique de classe et de la réflectivité. Ainsi, les graphes d'appels statiques de ces deux applications sont incomplets et conduisent à l'identification de mauvais composants. Par exemple, il y a 40 occurrences de la méthode `forName(String)` dans le code de PDFsam ce qui fait que son graphe d'appels statique est plus petit que son graphe d'appels dynamique (voir tableau 8.1.2).

Dans le cas de l'application CoCoME, les résultats obtenus avec le graphe d'appels statique sont moins bons. Mais, au vu des résultats obtenus pour la question **Q1**, il est difficile d'affirmer, dans ce cas, que les données dynamiques sont meilleurs aux données statiques.

Pour l'application JEval, il y a très peu de différences entre les résultats obtenus avec des données statiques et dynamiques. En effet, JEval n'utilise pas le chargement dynamique de classe ni la réflectivité, et utilise peu le polymorphisme. Ainsi, il n'y a que peu de différence entre le graphe d'appels dynamique et le graphe d'appels statique, et par la même entre les solutions identifiées avec ces deux graphes d'appels.

La question **Q2** n'est appliquée qu'à l'interprète Logo. Les résultats du tableau 8.III montrent clairement que les données dynamiques fournissent des solutions de meilleure qualité. Ceci est confirmé par une analyse plus précise des solutions (voir tableau 8.IV). En effet, dans l'échantillon des solutions choisis au hasard (20%) parmi les solutions obtenues avec des données statiques, aucune ne peut être considérée comme bonne.

8.4 Conclusion

Dans cette étude de cas, réalisée sur quatre applications, j'ai validé le processus d'identification de composant multi-criteres et multi-objectifs. Pour cela, j'ai montré que ce processus fournit des solutions proches de celles attendue par le concepteur. Il est sûrement possible d'améliorer les résultats en ajoutant de nouveaux critères tels que la cohésion sémantique.

J'ai aussi validé le choix multi-objectifs sur l'interprète Logo : ce processus fournit plusieurs bonnes décompositions de l'application cible. Il est donc intéressant de fournir un ensemble de solutions à l'utilisateur de cette approche.

Enfin, cette étude de cas a montré que dans le cas où les applications utilisent des mécanismes avancés du paradigme objet, tel que le chargement dynamique de classe, les graphes d'appels obtenus avec une analyse dynamique donnent de bien meilleurs résultats que ceux qui découle de l'utilisation de graphes d'appels obtenus statiquement.

CHAPITRE 9

EXEMPLE DE PROJECTION DE L'ABC VERS UN MODÈLE CONCRET

Dans ce chapitre, je montre la faisabilité de la réingénierie d'une application OO en application à base de composants suivant une architecture à base de composants. Ce travaille a été publié dans [ASSF11].

La restructuration de l'application OO cible en application à base de composants est effectuée en deux étapes. Premièrement, en utilisant des concepts du paradigme objet, j'implémente les interfaces requises et fournies des composants. Cela est présenté dans la section 9.1. Deuxièmement, les composants identifiés sont projetés vers un modèle de composants concret. Pour cette démonstration, j'ai choisi d'utiliser le modèle de composants OSGi [All09]. Cela est présenté dans la section 9.2.

Enfin, dans la section 9.3, je donne un exemple de cette réingénierie sur l'interprète Logo à partir d'une ABC identifiée avec le processus présenté dans cette thèse. L'identification de l'ABC ainsi que sa projection vers OSGi sont présentés.

9.1 Implémentation des interfaces

L'architecture à base de composant identifiée avec le processus présenté dans la partie III est une architecture composée d'éléments abstraits. Les interfaces des composants n'ont pas d'existence propre et sont seulement des éléments déduits de l'application OO cible. Pour rendre ces composants opérationnels, il est nécessaire de décrire et d'implémenter un mécanisme qui permet aux interfaces requises et fournies de travailler avec les classes qui les fournissent ou les requièrent.

9.1.1 Construction des interfaces fournies

Afin de se conformer au paradigme composant, seuls les services (méthodes) présents dans les interfaces fournies doivent être accessibles depuis les autres composantes de l'architecture, et cela seulement à travers ces interfaces. Pour atteindre cet objectif,

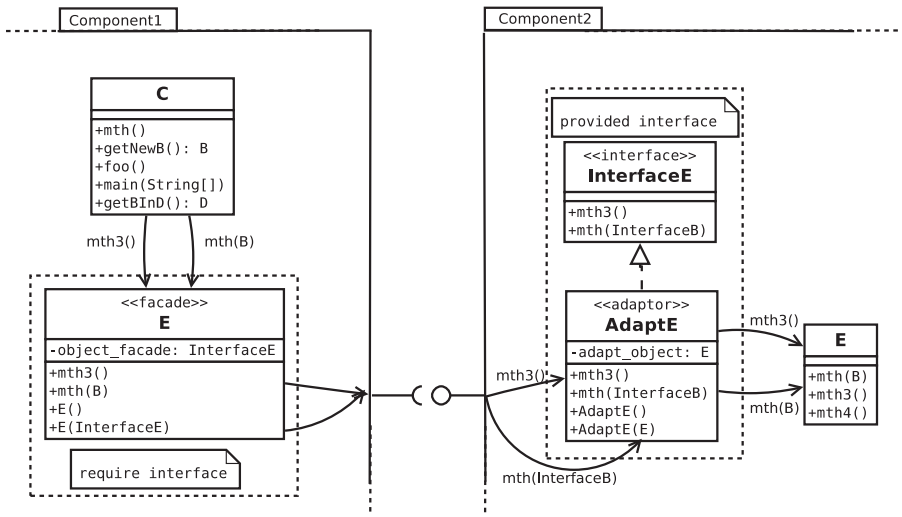


Figure 9.1 – Implémentation d'une interface avec des patrons de conceptions

j'ai choisi de ne pas modifier les classes existantes afin de ne pas compromettre leur cohérence interne et par la même compromettre l'application. Ainsi, j'ai décidé d'utiliser le patron de conception *adaptateur* [GHJV95]. Ceci est illustré par la figure 9.1. L'interface fournie `InterfaceE` du composant `Component2` est implémentée par l'adaptateur `adaptE`. Ce dernier adapte les services (méthodes) de la classe `E` (qui les implémente réellement) à l'interface fournie `interfaceE`. Le code de cette classe est donné ci-dessous :

```
class AdaptE implements InterfaceE {
  private E adapt_object;

  public AdaptE(){
    adapt_object = new E();
  }
  public void mth3(){ // delegation
    adapt_object.mth3();
  }
  public void mth(InterfaceB ib){
    // gestion des objets partagés
```

```
    B b = unWrapB(ib);
    adapt_object.mth3(b);
   }
}
```

La méthode `mth` reçoit en paramètre un objet de type `InterfaceB`. En réalité dans l'application OO originale, le paramètre est de type `B`. En effet, quelle que soit la localisation de la classe `B` (`Component2` dans l'exemple), dans la nouvelle application, les composants s'échangent uniquement des objets dont le type est défini par des interfaces fournies. Ainsi, avant d'être transmis comme paramètres, les objets sont encapsulés dans un type correspondant à leur interface fournie (voir paragraphe suivant). A leur réception les objets sont décapsulés afin d'obtenir leur type correct. Par exemple, cela est fait dans la méthode `mth` avec l'instruction `B b = unWrapB(ib)`, la classe `B` appartenant au composant `Component2` (voir figure 6.10b).

9.1.2 Construction des interfaces requises

Quand un composant a une interface requise, cela signifie qu'au moins une de ses classes utilise un service fourni par une classe située dans un autre composant. Plus précisément, celle-ci utilise un sous-ensemble des méthodes de la classe extérieur au composant. Ce sous-ensemble est représenté par une interface fournie dans le composant contenant cette classe. Pour rester cohérent avec le paradigme composant et permettre ainsi aux composants de ne voir que les services déclarés par l'interface fournie de l'autre composant, j'utilise le patron de conception *façade* [GHJV95] pour représenter une interface requise. Comme le montre la figure 9.1, chaque classe extérieure au composant et utilisée par celui-ci est remplacée par une classe du même nom qui agit comme une façade. Le code de la classe `E` du composant `Component1` est donné ci-dessous :

```
class E {
  private InterfaceE facade_object;
```

```
  public E() {
    facade_object = new AdaptE();
  }
  public void mth3(){
    facade_object.mth3();
  }
  public void mth(B b){
    InterfaceB ib = wrapB(b);
    facade_object.mth(ib);
  }
}
```

La classe `E`, du composant `Component1`, est une façade pour accéder aux services fournis par `InterfaceE` du composant `component2`. Elle garde le même nom (`E`) que la classe située dans `component2`, afin d'éviter de modifier le code des classes de `Component1`. Cette classe redirige les appels quelle reçoit vers un objet qui implémente l'interface `InterfaceE`. De plus, quand des objets sont passés en paramètres, la classe façade les encapsule dans un type connu par les autres composants (leur interface fournie correspondante). Dans l'exemple ci-dessus, ceci est obtenu par `wrapB()` dans la méthode `mth()`.

Les méthodes `wrap()` (classe façade) et la méthode `unwrap()` (classe adaptateur) forment ensemble un mécanisme qui assure que seuls les objets avec un type "public" (un type correspondant à une interface fournie) peuvent être échangés entre les composants. Par ailleurs, les classes qui utilisent des objets de type `E` peuvent également demander la création. Dans ce cas, le constructeur de la classe façade (voir ci-dessus le constructeur `E()`), qui sera appelé par ces classes, redirige la demande de création à l'adaptateur de la classe `E`, qui se trouve dans un autre composant (`Component2`).

9.2 Projection de l'architecture vers le modèle OSGi

Une fois que l'ABC de l'application cible est identifiée et que ses interfaces sont construites, il ne reste plus qu'à projeter les composants dans un modèle concret de composant pour obtenir l'application à base de composants. Pour illustrer cela, j'ai choisi d'utiliser le modèle de composant OSGi [All09]. Dans le reste de cette section, ce déploiement est présenté.

9.2.1 Creation des bundles

Dans le framework OSGi, un composant (appelé un bundle) est un ensemble de classes organisées en packages, qui par défaut ne sont pas visibles depuis l'extérieur du bundle. Le *manifest* du bundle permet d'exporter des packages : les classes et interfaces dans ces paquets exportés deviennent visibles à l'extérieur du bundle. Ainsi, ils agissent comme des interfaces fournies. De même, il est possible d'indiquer les packages dont le bundle a besoin pour fonctionner. Par conséquent, les classes et interfaces de ces packages jouent le rôle d'interfaces requises.

Pour exporter les interfaces fournies de nos composants à travers le *manifest*, ces interfaces sont placés dans des packages spécifiques. De même, les interfaces requises sont indiquées dans le *manifest* par l'importation des packages les contenant. En effet, ces interfaces requises sont forcément exportées par d'autres composants.

Par exemple, le bundle de la Figure 9.2 contient une interface fournie `InterfaceC` qui se trouve dans le package `interface_comp_2`. De plus, ce bundle a besoin des interfaces `InterfaceA` et `InterfaceE` du package `interface_comp_1`. Tout cela est spécifié dans le *manifest* comme suit :

```
Import-Package: interface_comp_2
Export-Package: interface_comp_1
```

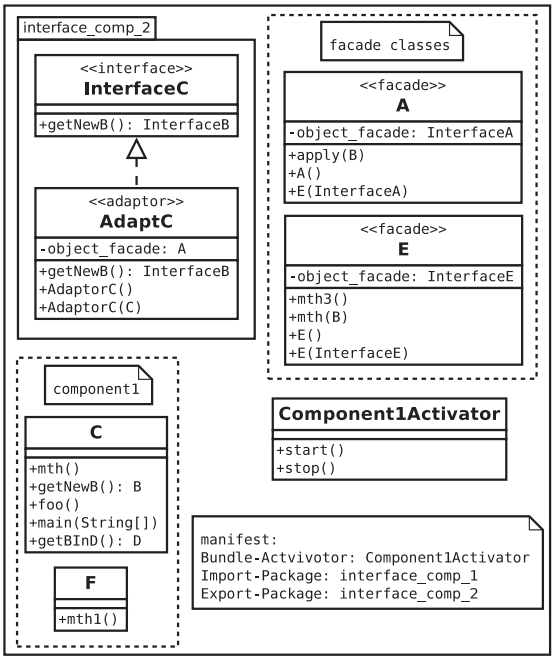


Figure 9.2 – Exemple de bundles

9.2.2 Gestion des activateurs

Une fois que la réingénierie de application orientée objet est effectuée, selon le modèle de composants concret, il ne reste plus qu'à prévoir un mécanisme pour lancer cette application. Le framework OSGi permet de spécifier des actions à réaliser au cours des différentes phases du cycle de vie des bundles (par exemple, *starting* ou *staping*) avec la classe `BundleActivator`. Ce mécanisme est utilisé pour lancer les nouvelles applications. Ainsi, pour chaque classe contenant un point d'entrée (la méthode `main()` en Java), je crée dans son bundle correspondant une sous-classe de la classe `BundleActivator` qui redéfinit la méthode `start(BundleContext)`. Ces sous-classes, appelées activateurs, définissent les actions du bundle durant ses différents cycles de vie. La méthode redéfinie est seulement utilisée pour appeler le point d'entrée initial (la méthode `main()`) de l'application originale. Le paramètre (`BundleContext`) de la methodes `start(BundleContext)` contient entre autres, le paramètre de la méthode `main()`. Si les classes d'un bundle contiennent plusieurs points d'entrée, le concepteur doit en choisir un. L'activateur du bundle est définie comme suit :

```
Bundle-Activator:
  activator.Component1Activator
```

Enfin, pour construire un bundle OSGi, les classes et les interfaces d'un composant, ses activateurs et son *manifest* sont archivés dans un fichier jar. Par exemple, la figure 9.2 montre le composant `Component1` structuré comme un bundle. Ce bundle se compose de classes `C` et `F`, son unique interface fournie (`InterfaceC`, son adaptateur `AdaptC`), et ses classes *facade* (`A` et `E`). Comme ce composant contient une classe avec un point d'entrée (la methode `main()` de la classe `C`), la classe `Component1Activator` a été créée et ajoutée à l'ensemble.

9.3 Exemple complet de réingénierie

Ce processus de projection d'une application OO le modèle de composant OSGi est testé sur l'interprète Logo. Le processus complet est presenté ci-dessous.

9.3.1 Identification des composants

Étant donné que les étapes 1 et 2 de l'identification d'une ABC ont été réalisées dans le chapitre 8 pour valider le processus d'identification des composants, elles ne sont pas détaillées ici. Les composants utilisés pour la suite du processus de réingénierie sont les composants référence de l'interprète Logo.

9.3.2 Identification des interfaces

A partir des composants de référence et d'un graphe d'appels, les interfaces sont identifiées. Ce graphe d'appels est obtenu en fusionnant les graphes d'appels dynamique et statique utilisés dans le chapitre 8 pour identifier les composants de l'interprète Logo. Durant cette étape, les composants ont été nommés manuellement en fonction de leurs interfaces fournies. Par exemple, le composant de la figure 9.3 fournit 4 interfaces : `ITrans`, `IUnknowFctException`, `IParseException` et `ITokenMgrError`. L'interface `ITrans` fournit des services qui permettent d'utiliser le parseur du langage

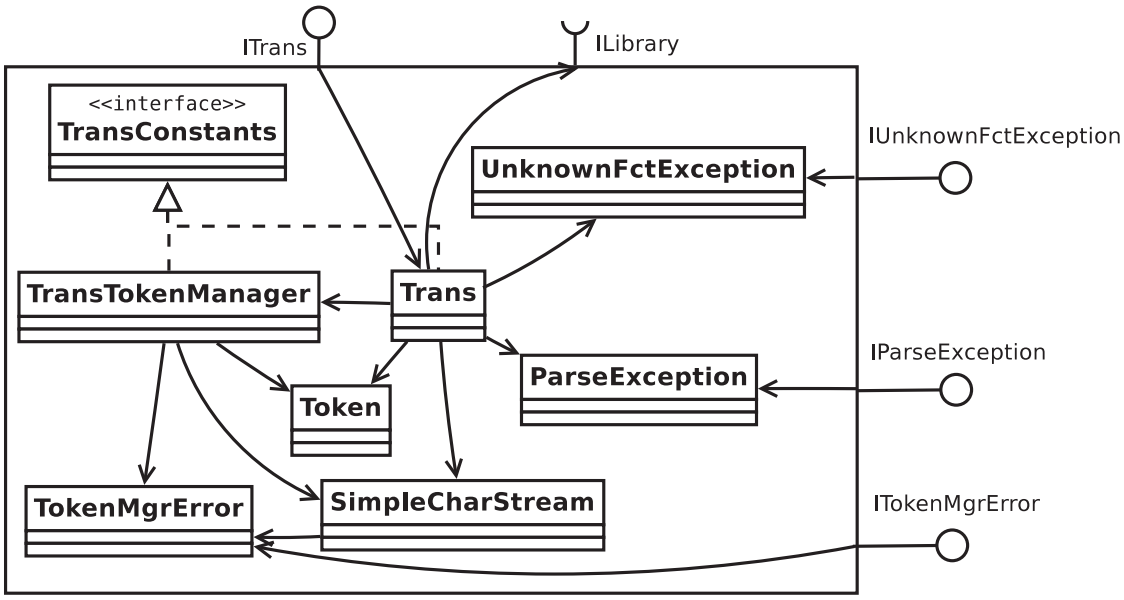


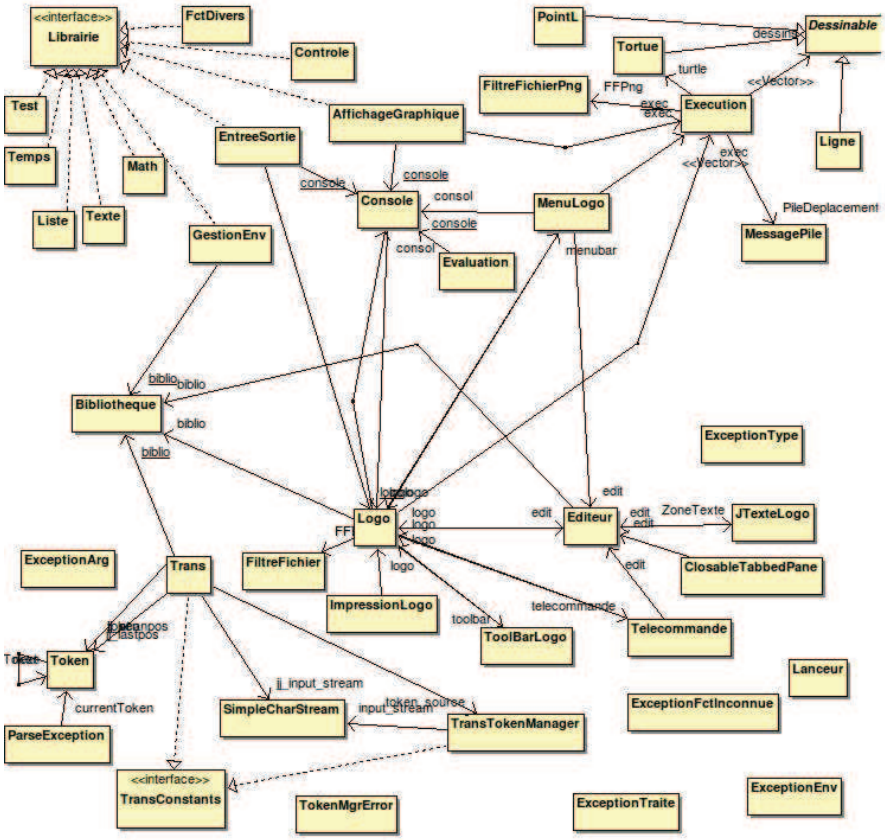
Figure 9.3 – Le composant `Parser` de l'architecture de la figure 9.4b

logo. Les trois autres interfaces fournissent des services pour gérer les erreurs du parseur. Ainsi, ces interfaces du composant, toutes liées au parseur, sont cohérentes entre elles. Ce composant est nommé `Parsor` .

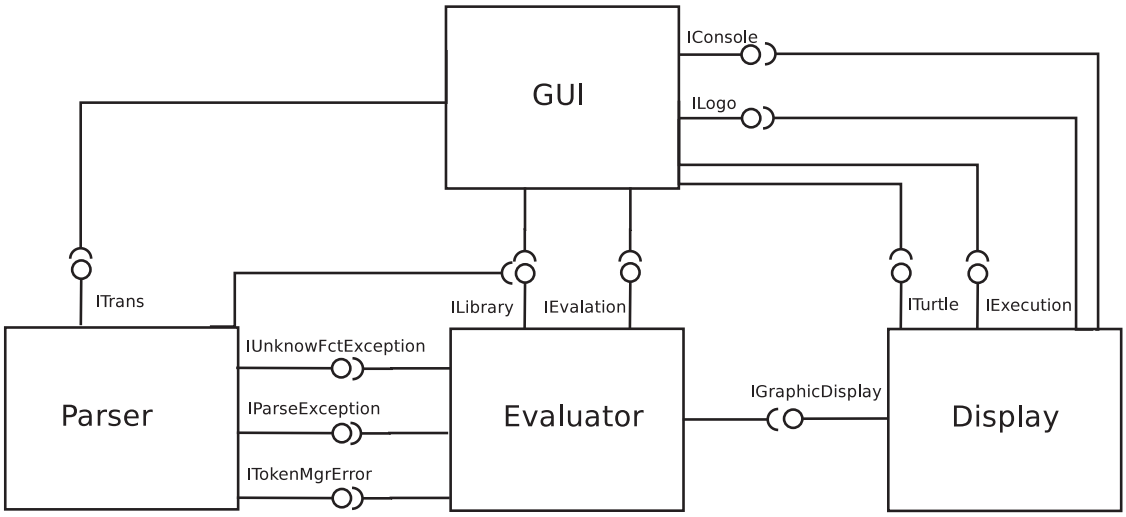
L'identification des interfaces fournies du composant `Display` est un exemple de la nécessité d'un graphe d'appels construit à l'aide de donnée statique et dynamique. En effet, dans la classe `Evaluation` du composant `Evaluator` la méthode `eval` utilise l'invocation de méthode dynamique pour appeler les méthodes de l'interprète Logo qui implémente les diverses fonctions du langage Logo. Cela est fait comme suit :

```
Object eval(List<Object> instr) {
...
   try {
       Method  mth = ((Method)instr.get(1)));
       return mth.invoke(null,v,env);
   }
...
}
```

Avec une analyse de type (analyse statique), il est impossible de déterminer les cibles



(a) L'architecture objet



(b) L'architecture à base de composants

Figure 9.4 – Architecture de l'interprète Logo

de la méthode `invoke` et ainsi le graphe d'appels sera incomplet. Or les méthodes fournissant les fonctions graphiques du langage Logo sont implémentées par la classe `GraphicDisplay` du composant `Display`, et ces méthodes sont utilisées par la classe `Evaluation`. Ainsi, sans les appels de méthodes obtenue dynamiquement à l'aide des cas d'utilisation, l'interface `IGraphicDisplay` ne pourrait pas être identifiée.

9.3.3 Construction des bundles

Les interfaces sont instanciées et les composants sont paquetagés sous forme de bundles OSGi, le tout de façon automatique en suivant l'approche décrite dans les sections 9.1 et 9.2. Le composant GUI contient le point d'entrée de l'application.

Afin de vérifier si la nouvelle application fournie les mêmes fonctionnalités que l'originale, l'ensemble des cas utilisation qui ont permis de tracer l'application originale sont joués dans la nouvelle application. Tous les cas d'utilisation sont exécutés correctement. De plus, malgré la double indirection d'appel pour les services entre deux composants, je n'est pas noté de ralentissement par rapport à l'application originale.

9.4 Conclusion

Dans ce chapitre j'ai montré qu'il est possible de projeter une application OO vers un modèle de composants concret en suivant une ABC. Pour cela j'ai proposé un processus de projection d'une ABC de l'application cible vers OSGi puis testé cette projection sur une application.

Cinquième partie

Conclusion

CHAPITRE 10

BILAN

La principale contribution de cette thèse est un processus semi-automatique d'identification d'une architecture à base de composants dans une application orientée objets. L'ABC obtenue a pour but l'aide à la compréhension de l'application originale. De plus, cette ABC peut servir de modèle pour une restructuration complète de l'application OO vers une application à base de composants.

Avant de définir le processus, un état de l'art a été réalisé afin d'identifier les problèmes non résolus par les approches existantes et de déterminer les points sur lesquels travailler. A partir de cela, j'ai proposé un processus semi-automatique, global, basé sur un modèle simple de composants. L'ensemble des composants identifiés (une partition de l'ensemble des classes de l'application cible) doit respecter les propriétés de cohérence et de complétude. Pour l'étape d'identification des composants, trois méthodes ont été explorées successivement avant d'obtenir une méthode pleinement satisfaisante. Vis à vis des approches publiées dans la littérature, les principaux apports sont les suivants :

- Un processus principalement guidé par des données issue d'une analyse dynamique.
- Une identification des composants multi-critères et multi-objectifs.
- Un processus qui permet de projeter l'ABC obtenue sur un modèle de composants concret.

Dans les sections suivantes, je vais faire un bilan rapide de ces différent apport.

10.1 Identification basée sur l'analyse dynamique

Ce processus d'identification est guidé par les relations et dépendances entre classes. Celles-ci sont principalement obtenues avec une analyse dynamique et non statique. En effet, comme cela a été montre dans cette thèse, les dépendances obtenues statiquement

conduisent à une solution conforme à la conception orientée objet de l'application. Or la vue composant de l'application cible peut être différente de la conception orientée objet. De plus, les mécanismes des langages-objets, tels que le chargement dynamique de classes, sont difficiles à simuler avec une analyse statique. Ainsi, l'analyse rencontre des problèmes pour déterminer correctement les dépendances issues de tels mécanismes. Cela a été montré dans le cas du calcul de la métrique de couplage CBO avec cinq algorithmes de construction de graphe d'appels.

Ainsi, j'ai proposé dans le cadre de cette thèse un processus guidé principalement par des dépendances obtenues avec une analyse dynamique. Dans ce cas, les dépendances entre classes sont capturées sous la forme de trace d'exécution, ces traces d'exécution étant elles-mêmes obtenues en exécutant tous les cas d'utilisations de l'application cible. L'identification des composants est vue comme le regroupement des classes présentées dans les traces d'exécution conformément à leurs interactions dans les cas d'utilisations. L'idée derrière cette approche est de regrouper les classes selon leurs dépendances fonctionnelles. Ceci est conforme avec la définition d'un composant : une entité fournissant une fonctionnalité.

Hélas, si le jeu de cas d'utilisation n'est pas complet, il est difficile de garantir la couverture de toutes les classes de l'application cible et de leurs inter-dépendances avec les traces d'exécution obtenues à partir de ce jeu. Or ce processus d'identification est global : toutes les classes de l'application doivent être assignées à un composant. Ainsi, j'ai proposé de compléter les dépendances obtenues dynamiquement avec des dépendances obtenues avec une analyse statique, si cela est nécessaire.

10.2 Processus multi-objectifs

Pour l'étape d'identification des composants, qui est au coeur du processus d'identification d'une ABC, j'ai exploré successivement trois méthodes. Chaque nouvelle méthode venant résoudre des problèmes identifiés durant la validation de la méthode précédente.

La première approche que j'ai étudiée utilise un treillis de Galois pour identifier les groupes de classes en relation directe. La seconde méthode utilise deux méta-heuristiques afin de partitionner les classes de l'application cible en fonction de leur proximité dans les traces d'exécutions. L'évaluation de ces deux méthodes m'a mené à la conclusion suivante : le couplage et la cohésion, comme uniques critères utilisés pour évaluer une solution, ne sont pas suffisants pour identifier les composants d'une application. D'autres critères se sont avérés nécessaires.

Ainsi, la troisième méthode redéfinit le problème d'identification d'une ABC en problème multi-critères et multi-objectifs basé sur le principe d'optimun de Pareto. La reformulation en problème multi-objectifs a été motivée par deux raisons. Premièrement, dans un problème multi-critères, agréger les différents critères dans une fonction d'évaluation est toujours une tâche délicate. Formuler le problème comme un problème multi-objectifs basé sur le principe d'optimun de Pareto permet de contourner ce problème. Secondement, il peut exister plusieurs bonnes décompositions de l'application cible. Et dans ce cas, il faut admettre que nous ne disposons toujours pas de tous les critères de sélection/évaluation. Ainsi, il est sage de considérer que seul le concepteur peut déterminer la solution parmi les meilleures qui lui convient le mieux.

Ainsi, cette méthode fournit un ensemble de bonnes solutions au concepteur qui choisit ensuite celle qui lui convient le mieux. L'ensemble de bonnes solutions est déterminé en utilisant un algorithme génétique multi-objectifs basé sur le principe d'optimun de Pareto. Cet algorithme utilise les critères suivant : le couplage, la cohésion, la granularité des composants et le nombre de cycles entre les composants. Cette méthode, la plus aboutie, a été évaluée en détail. Cette évaluation montre que ce processus fourni bien un ensemble de bonnes solutions. De plus, cet ensemble fournit des solutions très proches de celle attendue par le concepteur.

10.3 projection sur un modèle concret

Afin de pouvoir projeter les composants identifiés dans le modèle de composants utilisé dans cette thèse, et ainsi obtenir une ABC de l'application cible, j'ai proposé une

définition simple des interfaces des composants en usant des concepts orientés objets. Cette définition est : l'ensemble des services (méthodes) définis dans une classe appartient à la même interface. Cette définition est conforme au modèle de composants utilisé dans le cadre de cette thèse.

A partir de d'une ABC identifiée, j'ai pu valider un des objectifs secondaires de cette thèse : effectuer la réingénierie de l'application OO originale en application à base de composants. Pour cela, j'ai commencé par proposer une méthode pour implémenter les interfaces avec les patrons de conception *Adaptateur* et *Facade*. Ensuite, j'ai proposé une projection des composants de l'ABC cible et de ces interfaces dans le framework OSGi. Finalement, ce processus était validé avec succès sur une application.

10.4 Limitations

La première limitation de mon processus d'identification d'une ABC est liée à l'apport de dépendances statiques afin de compléter les dépendances dynamiques. En effet, si cet apport est trop grand (par exemple, si l'ensemble des cas d'utilisation utilisé dans la partie dynamique n'est pas complet) tout les avantages de l'analyse dynamique par rapport à l'analyse statique seront perdus. Or, durant l'évaluation du processus d'identification des composants, j'ai montré que les dépendances dynamiques donnent de meilleurs résultats.

La seconde limitation de l'étape d'identification des composants est liée à son aspect multi-objectifs. En effet, dans un problème multi-objectifs chaque ajout d'un nouveau critère augmente le nombre de bonnes solutions. Ainsi, si l'ensemble des critères est trop grand le concepteur peut être noyé sous une multitude de solutions lors de l'étape de sélection. L'ensemble des critères utilisés pour évaluer une solution doit être limité et soigneusement choisi.

Une autre limitation de mon approche est que les composants identifiés, puis projetés dans un modèle, le sont en fonction de l'application cible. Ainsi, ils peuvent être utilisés

pour restructurer l'application, mais je ne peux pas prétendre qu'ils sont réutilisables dans d'autres contextes. En effet, ces composants sont identifiés en fonction des cas d'utilisation de l'application cible. Or, ces cas d'utilisation sont définis dans le contexte précis de l'application. Ainsi, les composants qui en résultent sont liés à ce contexte. Il n'est pas exclu que ces composants soient réutilisables, mais je ne dispose pas de preuves empiriques pour affirmer cela.

CHAPITRE 11

PERSPECTIVES

Le processus d'identification présentées dans ce mémoire peut être amélioré sur différents points. Ces derniers constituent des pistes qui nécessitent plus de recherche et d'investigation. Ils portent sur toutes les étapes du processus.

11.1 Définition du problème

Pour l'instant, l'espace de recherche correspond à l'ensemble des partitions des classes de l'application qui respectent les contraintes de cohérence et de complétude. Or, il est possible d'appliquer d'autres contraintes à cet espace afin de faire décroitre sa taille. Par exemple, il est clair que si une classe n'a aucun lien avec les autres classes de son composant, celle-ci n'a pas à appartenir au composant. Une contrainte qui interdit les solutions de ce type serait facile à mettre en place. De plus, si ces contraintes sont paramétrables et optionnelles, il est possible de fournir un nouveau moyen de contrôle sur les solutions attendues, tout en réduisant l'espace de recherche.

Avec les contraintes de cohésion et de complétude, je simplifie le problème en supposant qu'une classe participe obligatoirement (complétude) à une unique (cohésion) fonctionnalité de haut niveau. Or, il est possible qu'une classe ne participe à aucune fonctionnalité de haut niveau et ne soit que du code *glue* entre deux fonctionnalités. Transposer dans le paradigme composant, cela revient à dire qu'une classe n'appartient à aucun composant et joue le rôle de connecteur entre deux composants. Cela est en contradiction avec la contrainte de complétude.

Au contraire, une classe peut participer à deux fonctionnalités de haut niveau, ce qui est en contradiction avec la contrainte de cohérence.

Ainsi, il me semble que le processus d'identification d'une ABC doit évoluer pour supprimer les contraintes de cohésion et de complétude afin d'améliorer l'étape d'iden-

tification des composants. Mais ceci peut avoir une influence non négligeable sur les algorithmes utilisés jusqu'ici, qui eux reposent sur les contraintes de cohésion et de complétude.

11.2 Type de dépendances

Comme nous l'avons vu, les seules dépendances que je considère sont les appels de méthodes et les lectures/écritures d'attributs entre classes. Or, il est possible de capturer d'autres types de dépendances comme le type des paramètres et de retour dans un appel de méthode, ou encore traiter différemment l'appel de méthode lié à l'instanciation d'un objet. Avec ces connaissances supplémentaires, il est, par exemple, possible de déterminer si l'objet de type A, utilisé dans les méthodes de la classe B, a été créé directement, dans les dites méthodes, ou s'il a été passé comme paramètre ou utilisé comme valeur retour. Cette information me semble intéressante pour enrichir le processus d'identification des composants.

Il sera aussi intéressant de considérer les patrons de conception utilisés dans l'application cible. En effet, ceux-ci apportent un supplément de sémantique et peuvent nous éclairer sur les intentions du concepteur de l'application. Par exemple, dans le patron de conception *Proxy*, la classe qui implémente le proxy peut être vue comme un connecteur entre deux composants.

11.3 Critères d'évaluation

Le processus que j'ai présenté dans ce mémoire se base sur quatre critères pour évaluer une ABC. Ces quatre critères ont été sélectionnés manuellement après avoir passé en revue les différentes métriques de qualité existantes pour les packages et les composants. Or, il serait possible d'identifier plus finement les meilleures métriques à utiliser avec une analyse statistique. Cela peut être fait en appliquant un ensemble de métriques sur un ensemble d'applications OO dont nous possédons une ABC. Ensuite, une analyse (analyse en composantes principales, régression linéaire, ...) peut être ap-

pliquée aux résultats des métriques afin de déterminer le sous-ensemble de métriques qui explique le mieux les ABC de leur application.

11.4 Projection

Une autre amélioration possible de mon approche est liée à la définition des interfaces des composants. En effet, la définition utilisée dans ce mémoire est quelque peu naïve. Or, il est possible de capturer plus de sémantique sur l'ensemble de classes qui fournit ces interfaces pour définir une meilleure répartition des services dans les interfaces. Par exemple, regrouper dans une même interface deux services qui implémentent, tous deux, une méthode déclarée dans la même interface (au sens Java). Une autre possibilité serait de regrouper les services en fonction de leur similitude syntaxique, sans tenir compte du paradigme objet dans lequel ces services sont implémentés.

Enfin, il faudrait généraliser le processus de projection de l'ABC vers des modèles concrets. Pour cela, il faudrait proposer d'autres projections vers d'autres modèles concrets (par exemple Fractal) puis déterminer les éléments communs et réutilisables entre ces différentes projections.

11.5 Réutilisation des composants

Comme je l'ai signalé dans le bilan, je n'ai pas de preuves empirique que les composants identifiés sont réutilisables, tout du moins, tous les composants. Il serait intéressant de définir des critères pour qualifier leur réutilisation potentielle. Dans une seconde étape, il serait possible d'intégrer ces critères dans l'étape d'identification des composants afin d'obtenir des composants réutilisables.

De plus, il est possible d'identifier les composants d'une librairie, qui a déjà été pensée en terme de réutilisabilité, à partir des cas d'utilisation des différentes applications qui l'utilise. Ainsi, les composants identifiés à partir des classes de la librairie ne seraient pas liés au contexte d'une application précise, mais à plusieurs contextes

d’applications qui utilisent cette librairie de différentes façons. Les composants qui en résultent seront surement plus réutilisables.

BIBLIOGRAPHIE

[All09] OSGI Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.2. Technical report, OSGI Alliance, September 2009.

[ASS09] Simon Allier, Houari A. Sahraoui, and Salah Sadou. Identifying components in object-oriented programs using dynamic analysis and clustering. In *CASCON*, pages 136–148, New York, NY, USA, 2009. ACM.

[ASSF11] S. Allier, S. Sadou, H. Sahraoui, and R. Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on*, pages 214 –223, june 2011.

[ASSV10] Simon Allier, Houari A. Sahraoui, Salah Sadou, and Stéphane Vaucher. Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In *CBSE*, pages 216–231, 2010.

[AVDS10] S. Allier, S. Vaucher, B. Dufour, and H. Sahraoui. Deriving coupling metrics from call graphs. In *10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 43 –52, 2010.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[BCS04] E. Bruneton, T. Coupaye, and J. B. Stefanie. The Fractal Component Model — Specification. Technical Report 2.0-3, The ObjectWeb Consortium, February 2004.

[BDW97] L.C. Briand, J.W. Daly, and J. Wust. A unified framework for cohesion measurement in object-oriented systems. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 43 –53, nov 1997.

[BDW99] L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1) :91–121, jan/feb 1999.

[BR00] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution : a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96 : Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM.

[CCI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery : A taxonomy. *IEEE Softw.*, 7 :13–17, January 1990.

[CCM96] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Softw. Pract. Exper.*, 26(1) :25–48, 1996.

[Cha85] Ned Chapin. Software maintenance : A different view. *Managing Requirements Knowledge, International Workshop on*, 0 :507, 1985.

[CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6) :476–493, jun 1994.

[CKK01] Eun Sook Cho, Min Sun Kim, and Soo Dong Kim. Component metrics to measure component quality. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 419 – 426, dec. 2001.

[CKK08] Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. Reverse engineering software-models of component-based systems. In *Proceedings of*

the 2008 12th European Conference on Software Maintenance and Reengineering, pages 93–102, Washington, DC, USA, 2008. IEEE Computer Society.

[CKO00] David Corne, Joshua D. Knowles, and Martin J. Oates. The pareto envelope-based selection algorithm for multi-objective optimisation. In *PPSN*, volume 1917 of *Lecture Notes in Computer Science*, pages 839–848. Springer, 2000.

[CSTO08] Sylvain Chardigny, Abdelhak Seriai, Dalila Tamzalit, and Mourad Oussalah. Quality-driven extraction of a component-based aachitecture from an object-oriented system. In *CSMR*, pages 269–273, 2008.

[CY91] Peter Coad and Edward Yourdon. *Object-oriented analysis (2nd ed.)*. Yourdon Press, Upper Saddle River, NJ, USA, 1991.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95 : Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag.

[DPAM02] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm : Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2) :182 –197, apr 2002.

[EKS03] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Trans. Software Eng.*, 29(3) :210–224, 2003.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Professional*, 2(3), 2000.

[Fal98] Emanuel Falkenauer. *Genetic algorithm and grouping problems*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[FCDCXF05] Meng Fan-Chao, Zhan Den-Chen, and Xu Xiao-Fei. Business component identification of enterprise information system : a hierarchical clustering method. In *ICEBE*, pages 473–480, 2005.

[FF93] C.M. Fonseca and P.J. Fleming. Genetic algorithms for multiobjective optimization : Formulation, discussion and generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416–423, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[Gar00] David Garlan. Software architecture : a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 91–101, New York, NY, USA, 2000. ACM.

[GD05] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR*, pages 314–323. IEEE Computer Society, 2005.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *OOPSLA '97 : Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 108–124, New York, NY, USA, 1997. ACM.

[GG03] N. S. Gill and P. S. Grover. Component-based measurement : few useful guidelines. *SIGSOFT Softw. Eng. Notes*, 28 :4–4, November 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, CA, 1979.

[GK95] H. Gall and R. Klosch. Finding objects in procedural programs : an alternative approach. In *WCRE*, pages 208–216, 1995.

[GMM+95] Robert Godin, Guy W. Mineau, Rokia Missaoui, Marc St-Germain, and Najib Faraj. Applying concept formation methods to software reuse. *International Journal of Software Engineering and Knowledge Engineering*, 5(1) :119–142, 1995.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[Gro06] Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006.

[HHN+94] Jeffrey Horn, Jeffrey Horn, Nicholas Nafpliotis, Nicholas Nafpliotis, David E. Goldberg, and David E. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *In Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, pages 82–87, 1994.

[HKW+08] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. The common component modeling example. chapter CoCoME - The Common Component Modeling Example. 2008.

[Hol75] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[JCIR01] Hemant Jain, Naresh Chalimeda, Navin Ivaturi, and Balarama Reddy. Business component identification - a formal approach. In *EDOC*, pages 183–187, Washington, DC, USA, 2001. IEEE Computer Society.

[JKL03] Yoon-Jung Jang, Eun-Young Kim, and Kyung-Whan Lee. Object-oriented component identification method using the affinity analysis technique. 2817 :317–321, 2003.

[KC04] Soo Dong Kim and Soo Ho Chang. A systematic method to identify software components. In *APSEC*, pages 538–545, Washington, DC, USA, 2004. IEEE Computer Society.

[KCLW08] Vincent Kelner, Florin Capitanescu, Olivier Léonard, and Louis Wehenkel. A hybrid optimization technique coupling an evolutionary and a local search algorithm. *J. Comput. Appl. Math.*, 215(2) :448–456, 2008.

[KE00] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 201 –210, 2000.

[KS08] Latika Kharb and Rajender Singh. Complexity metrics for component-oriented software systems. *SIGSOFT Softw. Eng. Notes*, 33 :4 :1–4 :3, March 2008.

[Lak96] John Lakos. *Large-scale C++ software design*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[LB85] M.M. Lehman and L. Belady. Program evolution : Process of software change. *London : Academic Press.*, 1985.

[LHJ10] William B. Langdon, Mark Harman, and Yue Jia. Efficient multi-objective higher order mutation testing with genetic programming. *J. Syst. Softw.*, 83 :2416–2430, December 2010.

[LHM07] Kiran Lakhotia, Mark Harman, and Phil McMinn. A multi-objective approach to search-based test data generation. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, GECCO '07, pages 1098–1105, New York, NY, USA, 2007. ACM.

[LLL08] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *ISSTA '08 : Proceedings of the 2008 international symposium on Software testing and analysis*, pages 131–142, New York, NY, USA, 2008. ACM.

[LMPR07] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *ASE*, pages 234–243, 2007.

[LNH07] V. Lakshmi Narasimhan and B. Hendradjaya. Some theoretical considerations for a suite of metrics for the integration of software components. *Inf. Sci.*, 177 :844–864, February 2007.

[LS] Bennet P. Lientz and Burton E. Swanson. *Software Maintenance Management : A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley.

[LS81] Bennett P. Lientz and E. Burton Swanson. Problems in application software maintenance. *Communiactions of the ACM*, 24(11), 1981.

[LSK+01] Jong Kook Lee, Seung Jae Seung, Soo Dong Kim, Woo Hyun, and Dong Han Han. Component identification method with coupling and cohesion. In *APSEC*, pages 79–86, Washington, DC, USA, 2001. IEEE Computer Society.

[LV01] D.H. Lorenz and J. Vlissides. Designing components versus objects : a transformational approach. In *ICSE*, pages 253–263, May 2001.

[LW90] S.-S. Liu and N. Wilde. Identifying objects in a conventional procedural language : an example of data design recovery. In *ICSM*, pages 266–271, 1990.

[Mar] Robert C. Martin. Design Principles and Design Patterns. Technical report.

[MB07] Onaiza Maqbool and Haroon Babri. Hierarchical clustering for software architecture recovery. *IEEE Trans. Softw. Eng.*, 33(11) :759–780, 2007.

[McK84] J.R. McKee. Maintenance as function of design. In *AFIPS National Computer Conference*, pages 187–193, 1984.

[Mey95] Bertrand Meyer. *Object Success. A Manager's Guide to Object Orientation, Its Impact on the Corporation and its Use for Reengineering the Software Process*. Object-Oriented Series. Prentice-Hall, Englewood Cliffs, New Jersy 07632, 1995.

[MJ06] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Eng.*, 13(2) :225–256, 2006.

[MM06] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.*, 32(3) :193–208, 2006.

[MM08] Brian S. Mitchell and Spiros Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Comput.*, 12(1) :77–93, 2008.

[MRR+53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21 :1087–1092, 1953.

[NW78] Albert. Nijenhuis and Herbert S. Wilf. *Combinatorial algorithms for computers and calculators / Albert Nijenhuis and Herbert S. Wilf*. Academic Press, New York :, 2d ed. edition, 1978.

[OL96] Ibrahim Osman and Gilbert Laporte. Metaheuristics : A bibliography. *Annals of Operations Research*, 63(5) :511–623, October 1996.

[OMG05] OMG. *Unified Modeling Language Specification 2.0 : Superstructure*, 2005. OMG doc. formal/05-07-04.

[Par00] David Lorge Parnas. Requirements documentation : Why a formal basis is essential. In *Proceedings of the 4th International Conference on Requirements Engineering (ICRE'00)*, ICRE '00, pages 81–, Washington, DC, USA, 2000. IEEE Computer Society.

[PGM+07] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and VÃ¡clav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6) :420–432, 2007.

[PHLR09] Chiragkumar Patel, Abdelwahab Hamou-Lhadj, and Juergen Rilling. Software clustering using dynamic analysis and static dependencies. In *CSMR*, pages 27–36, Washington, DC, USA, 2009. IEEE Computer Society.

[RBK+07] Ralf H. Reussner, Steffen Becker, Heiko Koziolek, Jens Happe, Michael Kuperberg, and Klaus Krogmann. The Palladio Component Model. Interner Bericht 2007-21, Universität Karlsruhe (TH), 2007. October 2007.

[SG96] Mary Shaw and David Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *OOPSLA '00 : Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 264–280, New York, NY, USA, 2000. ACM.

[SMLD97] Houari A. Sahraoui, Walcélio Melo, Hakim Lounis, and Francois Du-

mont. Applying concept formation methods to object identification in procedural code. In *ASE*, pages 210–218, 1997.

[Sou98] Maria Joao Sousa. A survey on the software maintenance process. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 265–, 1998.

[SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. Modernizing legacy systems : Software technologies, engineering processes, and business practices. *SEI Series in Software Engineering*, 2003.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TH00] Vassilios Tzerpos and R. C. Holt. Acdc : An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 258–, Washington, DC, USA, 2000. IEEE Computer Society.

[TP00] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00 : Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, New York, NY, USA, 2000. ACM.

[VC02] Marcello Visconti and Curtis R. Cook. An overview of industrial software documentation practice. In *Proceedings of the XII International Conference of the Chilean Computer Science Society*, SCCC '02, pages 179–, Washington, DC, USA, 2002. IEEE Computer Society.

[vDB11] Markus von Detten and Steffen Becker. Combining clustering and pattern detection for the reengineering of component-based software systems. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM*

SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS, QoSA-ISARCS '11, pages 23–32, New York, NY, USA, 2011. ACM.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework : Is it feasible ? In *International Conference on Compiler Construction (CC)*, pages 18–34, 2000.

[WF05] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Sci. Comput. Program.*, 56(1-2) :99–116, 2005.

[WFP+92] Alexander L. Wolf, C Fl, Dewayne Perry, Dewayne E. Perry, and Er L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17 :40–52, 1992.

[WT03] Zhihua Wen and V. Tzerpos. An optimal algorithm for mojo distance. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 227 – 235, may 2003.

[WT04] Zhihua Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 194 – 203, june 2004.

[YH10] Shin Yoo and Mark Harman. Using hybrid algorithm for pareto efficient multi-objective test suite minimisation. *J. Syst. Softw.*, 83 :689–701, April 2010.

[ZD08] Andy Zaidman and Serge Demeyer. Automatic identification of key classes in a software system using webmining techniques. *J. Softw. Maint. Evol.*, 20(6) :387–417, 2008.

[ZT98] Eckart Zitzler and Lothar Thiele. An evolutionary algorithm for multiobjective optimization : The strength pareto approach. Technical Report 43,

Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Gloriastrasse 35, CH-8092 Zurich, Switzerland, 1998.